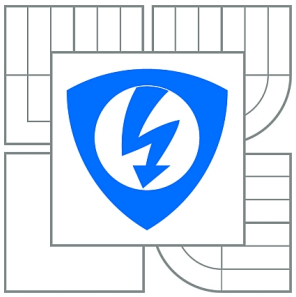
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ
ÚSTAV TELEKOMUNIKACÍ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION
DEPARTMENT OF TELECOMMUNICATIONS

VÝPOČETNÍ JEDNOTKY PROCESORŮ POSLEDNÍ GENERACE A JEJICH VYUŽITÍ

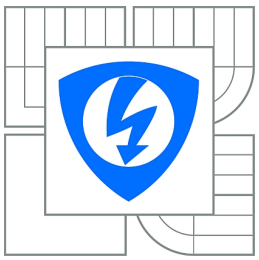
PROCESSING UNITS OF LAST GENERATION PROCESSORS AND THEIR UTILIZATION

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — SAMUEL ŠLENKER
AUTHOR

VEDOUCÍ PRÁCE — Ing. MIROSLAV BALÍK, Ph.D.
SUPERVISOR

BRNO 2015



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

Fakulta elektrotechniky a komunikačních technologií

Ústav telekomunikací

Bakalářská práce

bakalářský studijní obor
Teleinformatika

Student:	Samuel Šlenker	***ID:***	154887
Ročník:	3	***Akademický rok:***	2014/2015

NÁZEV TÉMATU:

Výpočetní jednotky procesorů poslední generace a jejich využití

POKYNY PRO VYPRACOVÁNÍ:

Proveďte podrobnou analýzu a následné porovnání starších a nových vektorových jednotek, které se používají v současných moderních procesorech. Soustřeďte se na porovnání možností vektorových jednotek SSE, AVX a FMA. Na konkrétních příkladech maticového a vektorového počtu uveďte přínosy AVX oproti SSE, dále přínosy tří- nebo čtyř-operandových operací podporovaných v rámci FMA. V úvahu vezměte také počty jader současných mikroprocesorů a počet současně pracujících vektorových jednotek na jádro. Výsledky zpracujte samostatně ve formě dvou teoretických úvodů k laboratorním úlohám. Připravte si vývojové prostředí pro vytváření dynamických knihoven s funkcemi, které budou tyto výpočetní jednotky využívat.

DOPORUČENÁ LITERATURA:

[1] Haswell New Instruction Descriptions [online], 2014. Dostupné z https://software.intel.com/en-us/blogs/2011/06/13/haswell-new-instruction-descriptions-now-available/
[2] Intel® Advanced Vector Extensions Programming Reference (319433-011) [online], 2014. Dostupné z https://software.intel.com/sites/default/files/m/f/7/c/36945

Termín zadání:	9.2.2015	***Termín odevzdání:***	2.6.2015

Vedoucí práce: Ing. Miroslav Balík, Ph.D.
Konzultanti bakalářské práce:

doc. Ing. Jiří Mišurec, CSc.
Předseda oborové rady

UPOZORNĚNÍ:

ABSTRAKT

Cieľom tejto práce bolo naštudovať a následne spracovať rozdiely medzi staršími inštrukčnými sadami a novšími inštrukčnými sadami, uviesť prínosy jednotlivých rozšírení, porovnať spôsoby výpočtov jednotlivých výpočtových SIMD jednotiek a porovnať ich implementáciu u firiem Intel a AMD. Súčasťou práce sú dva teoretické úvody k laboratórnym úlohám.

KĽÚČOVÉ SLOVÁ

SIMD, SSE, AVX, FMA, vektorové spracovanie dát, Intel, AMD

ABSTRACT

The aim of this thesis was to study and subsequently process the differences between the older instruction sets and newer instruction sets, to specify the benefits of the individual extensions, to compare the way of computations of the individual SIMD processing units and to compare the implementation of these processing units in Intel and AMD companies. Part of this work are two theoretical introductions to laboratory tasks.

KEYWORDS

SIMD, SSE, AVX, FMA, vector data processing, Intel, AMD

ŠLENKER, Samuel *Výpočetní jednotky procesorů poslední generace a jejich využití.*: bakalárska práca. Místo: Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav telekomunikací, Rok. 96 s. Vedúci práce bol Ing. Miroslav Balík, Ph.D.

PREHLÁSENIE

Prehlasujem, že som svoju bakalársku prácu na tému „Výpočetní jednotky procesorů poslední generace a jejich využití.“ vypracoval(a) samostatne pod vedením vedúceho bakalárskej práce, s využitím odbornej literatúry a ďalších informačných zdrojov, ktoré sú všetky citované v práci a uvedené v zozname literatúry na konci práce.

Ako autor(ka) uvedenej bakalárskej práce ďalej prehlasujem, že v súvislosti s vytvorením tejto bakalárskej práce som neporušil(a) autorské práva tretích osôb, najmä som nezasiahol(-la) nedovoleným spôsobom do cudzích autorských práv osobnostných a/alebo majetkových a som si plne vedomý(-á) následkov porušenia ustanovenia § 11 a nasledujúcich autorského zákona č. 121/2000 Sb., o práve autorskom, o právach súvisejúcich s právom autorským a o zmene niektorých zákonov (autorský zákon), v znení neskorších predpisov, vrátane možných trestnoprávnych dôsledkov vyplývajúcich z ustanovenia časti druhej, hlavy VI. diel 4 Trestného zákoníka č. 40/2009 Sb.

Místo ....

podpis autora(-ky)

POĎAKOVANIE

Rád by som poďakoval vedúcemu bakalárskej práce pánovi Ing. Miroslavovi Balíkovi, Ph.D. za odborné vedenie, konzultácie, trpezlivosť a podnetné návrhy k práci.

Místo ....

podpis autora(-ky)

Faculty of Electrical Engineering
and Communication
Brno University of Technology
Purkynova 118, CZ-61200 Brno
Czech Republic

http://www.six.feec.vutbr.cz

POĎAKOVANIE

Výskum popísaný v tejto bakalárskej práci bol realizovaný v laboratóriach podporených projektom SIX; registračné číslo CZ.1.05/2.1.00/03.0072, operačný program Výzkum a vývoj pro inovace.

Místo . . . . . .

. . . . . .

podpis autora(-ky)

EVROPSKÁ UNIE
EVROPSKÝ FOND PRO REGIONÁLNÍ ROZVOJ
INVESTICE DO VAŠÍ BUDOUCNOSTI

OBSAH

ZOZNAM OBRÁZKOV

ZOZNAM TABULIEK

ÚVOD

Vektorové výpočty vznikli najmä vďaka pokročilým operáciam s 3D grafikou vôbec. V tejto oblasti hrajú veľkú úlohu maticové a vektorové výpočty. Práve vďaka týmto nárokom sa začali už staršie výpočtové jednotky procesorov vyvíjať smerom vektorového spracovania. Vďaka vzniku vektorových výpočtových jednotiek sa značne urýchlili operáciami s maticami a vektormi, keďže vektorové výpočtové jednotky dokážu spracovať v jednom kroku viacero hodnôt.

Zameranie tejto práce je na vektorové výpočtové jednotky procesorov staršej generácie a ich následné porovnanie s vektorovými výpočtovými jednotkami novej generécie. V práci je uvedená história výpočtových systémov so zameraním na vývoj systému spracovania viacerých dát použitím jednej inštrukcie.

Prehľad výpočtových jednotiek staršej a novšej generácie poskytuje základné informácie o tom, ako tieto výpočtové jednotky pracujú, tj. s akými hodnotami sú dané výpočtové jednotky schopné pracovať, akým spôsobom a kam sa nahrávajú hodnoty, ktoré budú následne spracované či spôsob samotného spracovania dát. Z toho vyplunul prínos jednotlivých výpočtových jednotiek.

V práci je zahrnutá implementácia vektorových výpočtových jednotiek v mikroprocesoroch od firiem Intel a AMD. Práca poskytuje detailný pohľad na najnovšie mikroarchitektúry procesorov oboch firiem spolu so spätným pohľadom na predošlé mikroarchitektúry, v ktorých boli implementované staršie výpočtové jednotky.

Keďže výpočtové jednotky sa pri operáciach riadia príkazmi, tzv. inštrukciami, práca poskytuje zoznam inštrukčných súborov a samotných inštrukcií pridaných v jednotlivých inštrukčných sadách.

Samotné porovnanie vektorových výpočtových jednotiek procesorov je v práci realizované testovaním zdrojových kódov výpočtu matice násobenej vektorom, resp. vektoru násobeného maticou. Pri realizácii výpočtov sú využité vlastnosti týchto výpočtových jednotiek, ktoré viedli k optimalizácii zdrojových kódov pre dosiahnutie čo najrýchlejšieho času výpočtu.

1 HISTÓRIA A VLASTNOSTI VÝPOČTOVÝCH SYSTÉMOV

V roku 1966 M. J. Flynn systematicky roztriedil a popísal výpočtové systémy, ktoré sú rozdelené podľa počtu inštrukčných a dátových tokov spracovaných súčasne, t.j. v jednom okamihu. Tieto výpočtové systémy ostali ústredným bodom rozdelenia dodnes. M.J.Flynn ich označil nasledovne: SI (Single Instruction), MI (Multiple Instruction), SD (Single Data) a MD (Multiple Data). Z predchádzajúcich skratiek vznikli názvy pre výpočtové systémy SISD (Single Instruction-Single Data), SIMD (Single Instruction-Multiple Data), MISD (Multiple Instruction-Single Data) a MIMD (Multiple Instruction-Multiple Data) [1].

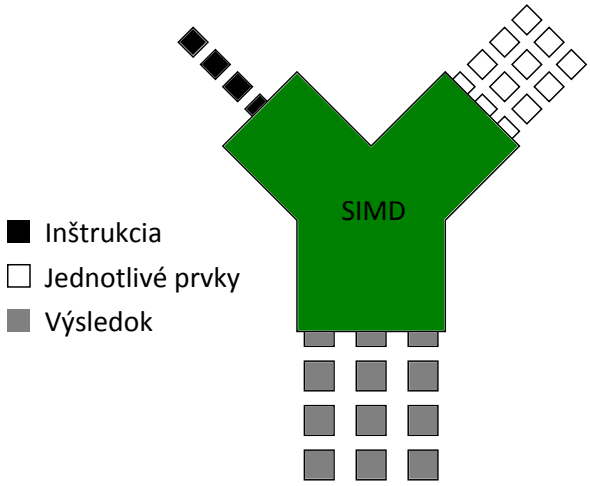
1.1 Výpočtové systémy

- SISD odkazuje na počítačovú architektúru, v ktorej jeden procesor (uniprocessor) vykoná jednu inštrukciu na dátach uložených v jednej pamäti.
- SIMD systém predstavuje vektorové operácie. Jedná sa o spracovanie viacerých dát jedinou inštrukciou v danom okamihu.
- MISD je typ paralelnej výpočtovej architektúry, kde viacero funkčných jednotiek vykonáva rôzne operácie na rovnakých dátach.
- MIMD používa niekoľko procesorov, ktoré pracujú asynchrónne a nezávisle. Rôzne procesory môžu kedykoľvek vykonávať rôzne inštrukcie na rôznych dátach.

1.2 Použitie systému SIMD

Prvá implementácia systému SIMD bola vo vektorových superpočítačoch v sedemdesiatych rokoch 20. storočia, ktoré dokázali pracovať s vektorom dát použitím jedinej inštrukcie. Tieto superpočítače boli charakteristické masívnym vektorovým spracovaním dát.

SIMD systémy v superpočítačoch po čase nahradili MIMD systémy a v osobných počítačoch, v ktorých sa spočiatku využíval systém SISD, sa začal používať systém SIMD. SIMD inštrukcie v súčasnosti používa väčšina jednotiek CPU (Central Processing Unit). Najväčší prínos technológie SIMD je v oblastiach multimédií, simulácií a iných aplikácií, ktoré sú založené na rozsiahlom vektorovom a maticovom výpočte. Pri týchto typoch výpočtov sa vykonáva rovnaká operácia na viacerých prvkoch. Spôsob, akým systém SIMD pracuje, zobrazuje obr. 1.1.



Obr. 1.1: Spracovanie dát v systéme SIMD

1.3 Vývoj SIMD technológie v osobných počítačoch

- Rok 1996 - prvá implementácia SIMD technológie - mikroprocesor Intel Pentium MMX prináša rozšírenie MMX (najčastejšie sa význam skratky definuje ako Multiple Math eXtension, avšak používajú sa aj názvy MultiMedia eXtension alebo Matrix Math eXtension).
- Rok 1999 - mikroprocesor Intel Pentium III prináša rozšírenie SSE (Streaming SIMD Extensions).
- Rok 2000 - mikroprocesor Intel Pentium IV prináša rozšírenie SSE2.
- Rok 2004 - verzia mikroprocesoru Intel Pentium IV s názvom Prescott prináša rozšírenie SSE3.
- Rok 2006 - mikroprocesor Intel Core 2 prináša rozšírenie SSSE3 (Suplemental Streaming SIMD Extensions).
- Rok 2008 - verzia mikroprocesoru Intel Core 2 s názvom Wolfdale prináša rozšírenie SSE4.1.
- Rok 2008 - mikroprocesor Intel s mikroarchitektúrou Nehalem prináša rozšírenie SSE4.2.
- Rok 2011 - mikroprocesor Intel s mikroarchitektúrou Sandy Bridge prináša rozšírenie AVX (Advanced Vector Extensions).
- Rok 2013 - mikroprocesor Intel s mikroarchitektúrou Haswell prináša rozšírenie AVX2 a FMA3 [2].

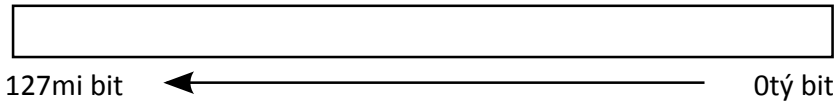
2 POPIS VÝPOČTOVÝCH JEDNOTIEK

2.1 Výpočtová jednotka SSE

Technológia SSE pridáva k predošlej technológii MMX niekoľko nových vlastností. Z pohľadu výpočtových jednotiek je najdôležitejšie pridanie schopnosti práce s FP operáciami s jednoduchou presnosťou (Single-Precision Floating-Point) a dvojitou presnosťou (Double-Precision Floating-Point) [2]. Tieto operácie zaisťujú inštrukčné súbory SSE (kap. 4.1) a SSE2 (kap 4.2), nové dátové registre (kap. 2.1.1) a nové dátové typy (kap. 2.1.2).

2.1.1 Dátové registre

Výpočtová jednotka SSE využíva pri výpočtoch dátové registre nazývané XMM. Počet týchto registrov je primárne daný režimom procesora. 64bitový operačný systém prepína po štarte procesor do tzv. dlhého režimu (Long Mode), zatiaľ čo 32bitový operačný systém nastavuje procesor do chráneného režimu. Chránený režim umožňuje použitie 8mich XMM registrov, dlhý režim 16tich XMM registrov. K týmto registrom sa dá pristupovať priamo, použitím ich mien xmm0-xmm15 a prístup k nim je nezávislý od ostatných registrov. Každý register má veľkosť 128 bitov (obr. 2.1).



Obr. 2.1: XMM register

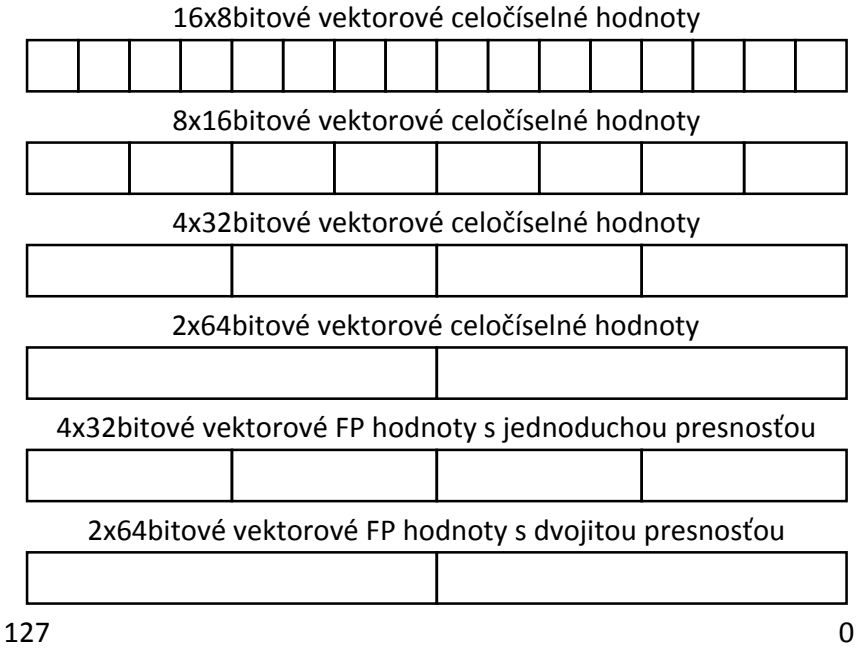
2.1.2 Dátové typy

Dátové typy predstavujú hodnoty, ktorými sa naplnia dátové registre. Keďže XMM registre majú šírku 128 bitov, tak aj dátové typy môžu naplniť register len do tejto šírky. Výpočtová jednotka SSE disponuje štyrmi dátovými typmi. Sú to

- zabalený dátový typ Packed Byte (16x8 bitov),
- zabalený dátový typ Packed Word (8x16 bitov),
- zabalený dátový typ Packed Doubleword (4x32 bitov),
- zabalený dátový typ Packed Quadword (2x64 bitov).

Pri numerických SIMD operáciach sú tieto dátové typy interpretované na základe typu hodnôt, ktoré nesú. Sú to (obr. 2.2)

- 16x8bitové vektorové celočíselné hodnoty (Packed Byte Integers),
- 8x16bitové vektorové celočíselné hodnoty (Packed Word Integers),
- 4x32bitové vektorové celočíselné hodnoty (Packed Doubleword Integers),
- 2x64bitové vektorové celočíselné hodnoty (Packed Quadword Integers),
- 4x32bitové vektorové FP hodnoty s jednoduchou presnosťou (Packed Single-Precision Floating-Point),
- 2x64bitové vektorové FP hodnoty s dvojitou presnosťou (Packed Double-Precision Floating-Point) [2].



Obr. 2.2: SSE - dátové typy

2.1.3 Spôsob výpočtu

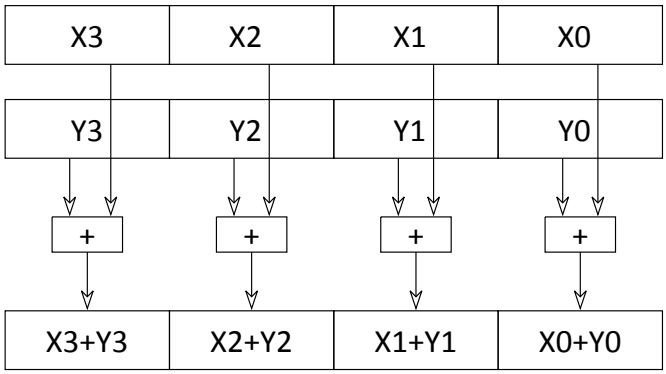
Výpočtové operácie sa vykonávajú pomocou špeciálnych príkazov nazývaných inštrukcie. Inštrukcia definuje operáciu a dátový typ. Za inštrukciou nasledujú operandy, na ktorých sa daná operácia vykoná. Syntax pri SSE dovoľuje len dva operandy - jeden zdrojový operand, a jeden operand, ktorý je zdrojový a zároveň aj cieľový. Napr. operácia priradenia

$$\vec{a} \leftarrow \vec{a} + \vec{b}$$

pri použití dátového typu Doubleword, v jazyku symbolických inštrukcií predstavovala inštrukcia PADDD (Packed Add Doubleword). Syntax inštrukcie by bol nasledovný

paddd xmm1, xmm2 .

Správanie tejto inštrukcie znázorňuje obr. 2.3.



Obr. 2.3: Vektorové sčítanie pomocou inštrukcie PADDD

Zdrojovými operandami sú oba registre XMM a zároveň cieľovým operandom je XMM register, ktorý je zapísaný ako prvý, teda XMM1. Výpočtová jednotka SSE teda pri operáciach používa tzv. deštruktívny operand.

2.2 Výpočtová jednotka AVX

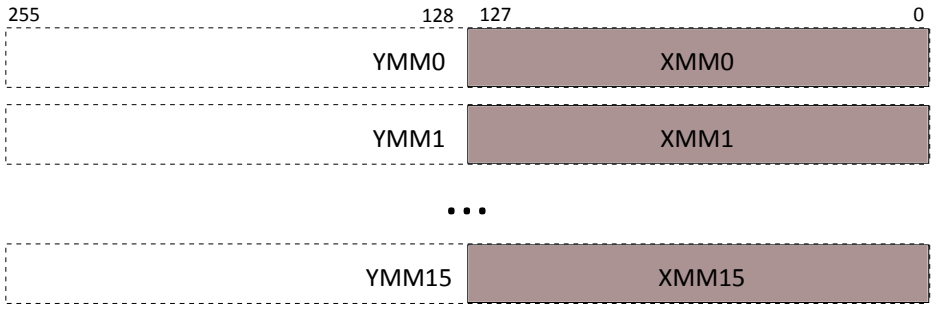
Technológia AVX predstavuje výrazné rozšírenie predošlej technológie SSE. Z pohľadu výpočtových jednotiek sú najvýznamnejšie tieto zmeny [2]:

- nové 256bitové registre YMM (kap. 2.2.1),
- nové dátové typy (kap. 2.2.2),
- inštrukčné súbory AVX (kap. 4.7), FMA3 (kap. 4.9) a AVX2 (kap. 4.8) poskytujú inštrukcie pre prácu s novými registrami a dátovými typmi,
- novú kódovaciu schému VEX (Vector Extension) (kap. 2.2.3).

2.2.1 Dátové registre

Výpočtová jednotka AVX prináša nové registre nazývané YMM. Počet registrov je opäť daný verziou operačného systému. Chránený režim umožňuje používať 8 YMM

registrov, dlhý režim 16 YMM registrov. YMM registre sa prekrývajú s XMM registrami. Tie tvoria dolnú polovicu bitov XMM registrov (obr. 2.4). Pri použití inštrukcií pracujúcich na XMM registroch sa horná polovica bitov YMM registrov vynuluje [2] [3].



Obr. 2.4: YMM registre

2.2.2 Dátové typy

K predošlým dátovým typom SSE postupne pribudli dátové typy rozšírené o dvojnásobnú veľkosť. Výpočtová jednotka AVX tak dokáže uložiť v jednom YMM registri

- 32x8bitové vektorové celočíselné hodnoty (Packed Byte Integers),
- 16x16bitové vektorové celočíselné hodnoty (Packed Word Integers),
- 8x32bitové vektorové celočíselné hodnoty (Packed Doubleword Integers),
- 4x64bitové vektorové celočíselné hodnoty (Packed Quadword Integers),
- 8x32bitové vektorové FP hodnoty s jednoduchou presnosťou (Packed Single-Precision Floating-Point),
- 4x64bitové vektorové FP hodnoty s dvojitou presnosťou (Packed Double-Precision Floating-Point) [3] [4].

2.2.3 Spôsob výpočtu

Problém deštruktívneho operandu vyriešil príchod technológie AVX. Pomocou kódovacej schémy VEX sú všetky nové inštrukcie a takmer všetky predošlé inštrukcie schopné pracovať s nedeštruktívnym operandom, pretože schéma VEX dovoľuje použitie jedného cieľového operandu a až dvoch zdrojových operandov. To znamená, že operácia priradenia z kap. 2.1.3 by pri využití výpočtovej jednotky AVX mala numerický zápis

$$\vec{c} \leftarrow \vec{a} + \vec{b}\,.$$

V jazyku symbolických inštrukcií by danú operáciu predstavovala inštrukcia VPADDD (Vex Packed Add Doubleword). Zápis tejto inštrukcie by vyzeral takto:

vpaddd xmm0, xmm1, xmm2 .

Písmeno „v“ na začiatku inštrukcie indikuje použité kódovanie pomocou schémy VEX. Z príkladu je zrejmé, že boli použité 3 operandy. Prvý operand je cieľový. Nasledujúce dva operandy sú zdrojové a určujú poradie pri vykonávaní inštrukcie. V tomto prípade sa hodnoty XMM1 vektorovo sčítajú s hodnotami XMM2 a výsledok sa uloží do XMM0.

S deštruktívnym operandom sa však stretneme aj pri technológií AVX. Výpočtová jednotka AVX v novších mikroarchitektúrach procesorov využíva inštrukčné súbory FMA3 a FMA4. Tieto prinášajú nové SIMD inštrukcie, ktoré dokážu vykonávať operácie násobenia a sčítania (zahŕňajúc operácie násobenia a odčítania alebo iných variácií) na vektorových alebo skalárnych dátových prvkoch v jednom kroku. Tým sa zlepšuje výpočtový výkon a presnosť výpočtov, pretože pri použití samostatných operácií násobenia a následného sčítania sa výsledné hodnoty zaokrúhľujú po každej operácií, tj. vykonajú sa dve zaokrúhlenia. Použitím FMA sa zaokrúhľuje až konečný výsledok a tým pádom FMA zmenšuje chyby po zaokrúhľovaní.

Zatiaľ, čo verzia FMA3 využíva deštruktívny operand, verzia FMA4 využíva nedeštruktívny operand. Numerické správanie FMA inštrukcií sa dá demonštrovať nasledovne s použitím FMA3 (2.1) alebo FMA4 (2.2)

$$\vec{a} \leftarrow \vec{a}.\vec{b} + \vec{c} \tag{2.1}$$

$$\vec{d} \leftarrow \vec{a}.\vec{b} + \vec{c} \tag{2.2}$$

Verzia FMA4 bola zrealizovaná skôr ako verzia FMA3, v roku 2011 firmou AMD v mikroarchitektúre Bulldozer. Verziu FMA3 začala firma AMD používať v mikroarchitektúre Piledriver a firma Intel v mikroarchitektúre Haswell. Firma Intel verziu FMA4 nevyužíva.

3 IMPLEMENTÁCIA VÝPOČTOVÝCH SIMD JEDNOTIEK

3.1 Firma Intel

3.1.1 Mikroarchitektúra Haswell

Haswell je v súčasnosti najnovšia mikroarchitektúra procesorov od firmy Intel, ktorá je dostupná na trhu. Je to prvá mikroarchitektúra od firmy Intel, ktorá poskytuje rozšírenia AVX2 a FMA3.

Výpočtové jednotky sú implementované v mikroprocesore. Počet výpočtových jednotiek je daný počtom jadier. Každé jadro mikroprocesoru obsahuje rezervačnú stanicu (Reservation Station), výstupné porty a výpočtové jednotky.

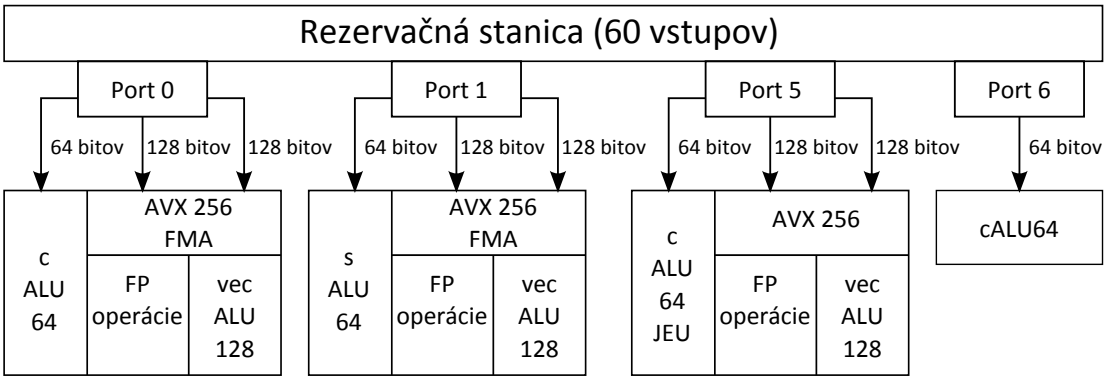
Rezervačná stanica je základom jednotky. Na rezervačnú stanicu sú pomocou zásobníkov pripojené výstupné porty. Zásobníky majú šírky 64 alebo 128 bitov. Na vstup rezervačnej stanice sú privádzané dekódované inštrukcie, pričom sú privádzané len tie mikrooperácie, ktoré majú status kompletnosti (sú im pridelené aktuálne hodnoty operandov). V rezervačnej stanici následne dochádza k rozdeľovaniu a priraďovaniu jednotlivých mikrooperácií k výstupným portom. Toto sa deje na základe statusu, či je potrebná výpočtová jednotka k dispozícii (či nie je použitá pre inú, práve vykonávanú mikrooperáciu) [5].

Mikroarchitektúra Haswell má k dispozícii celkom 8 portov označených Port 0 až Port 7:

- **Port 0** - pripojené jednotky cALU64 (complex Arithmetic Logic Unit 64), jednotka pre výpočet FP operácií (tzv. FPU - Floating-Point Unit) a jednotka vecALU128 (vector Arithmetic Logic Unit 128) alebo jednotky AVX 256 a FMA.
- **Port 1** - pripojené jednotky sALU64 (simple Arithmetic Logic Unit 64), jednotka FPU a jednotka vecALU128 alebo jednotky AVX 256 a FMA.
- **Port 2** - pripojená jednotka uAGU (universal Address Gerenation Unit).
- **Port 3** - pripojená jednotka uAGU.
- **Port 4** - pripojená jednotka pre uloženie dát (Store Data).
- **Port 5** - pripojené jednotky cALU64 alebo JEU (Jump Exexution Unit), jednotka FPU a jednotka vecALU128 alebo jednotky AVX 256 a FMA.
- **Port 6** - pripojená jednotka cALU64.
- **Port 7** - pripojená jednotka AGU.

Aj keď sa niektoré výpočtové jednotky na rôznych portoch nazývajú rovnako, operácie, ktoré poskytujú, nemusia byť rovnaké. Zoznam konkrétnych operácií na kon-

krétnych výpočtových portoch poskytuje tabuľka na konci kapitoly (tab. 3.1). Zjednodušená bloková schéma zapojenia je na obr. 3.1.



Obr. 3.1: Haswell - zjednodušená bloková schéma pripojenia výpočtových jednotiek k portom a rezervačnej stanici v jednom jadre

Technológie AVX, AVX2 a FMA3 poskytujú možnosť práce s 256bitovými hodnotami, spracovanými v jednom kroku. Z obrázku je zrejmé, že jednotky AVX 256 a FMA združujú šírky oboch 128bitových zásobníkov pre jednotky FPU a vecALU128, aby dokázali spracovať 256bitové operácie. Jednotky nedokážu pracovať súčasne. Buď pracuje jednotka AVX 256 alebo FMA, alebo jednotky vecALU128 a FPU. Rovnako na porte 5 buď pracuje jednotka cALU64 alebo jednotka JEU [5]. Všetky výpočtové jednotky sú špecializované pre rôzne operácie. Nasleduje všeobecný popis jednotlivých výpočtových jednotiek:

- Jednotka cALU64 slúži k výpočtom celočíselných operácií. Táto jednotka obsahuje posúvač.
- Jednotka sALU64 slúži k výpočtom celočíselných operácií. Táto jednotka posúvač neobsahuje.
- Jednotka FPU slúži k výpočtom operácií s rádovou plávajúcou čiarkou.
- Jednotky slúžiace k vektorovým SIMD výpočtom sú nazývané združene ako jednotka vecALU128.
- Jednotky AVX256 a FMA vykonávajú 256bitové vektorové SIMD výpočty.
- Jednotka uAGU slúži na výpočet adresy pre uloženie alebo načítania dát.
- Jednotka AGU slúži na výpočet adresy pre uloženie dát.
- Jednotka JEU je jednotka skokových operácií.

Kompletný zoznam konkrétnych výpočtových operácií na daných výstupných portoch poskytuje tabuľka 3.1 [6].

Tab. 3.1: Výpočtové operácie na výstupných portoch

Port	Operácie
0	aritmeticko-logické operácie, posun (shift), vektorový posun
0	FP násobenie a FMA operácie
0	celočíselné vektorové násobenie
0	vektorové logické operácie
0	vetvenie (branch)
0	delenie, odmocnenie
1	aritmeticko-logické operácie
1	FP sčítanie, FP násobenie a FMA operácie
1	komplexné celočíselné operácie (napr. skenovanie bitov)
1	celočíselné násobenie
1	vektorové logické operácie
2	výpočet adresy pre načítanie/uloženie dát
3	výpočet adresy pre načítanie/uloženie dát
4	uloženie dát (store data)
5	aritmeticko-logické operácie, posun (shift)
5	vektorové permutácie, skokové operácie
5	vektorové logické operácie
6	aritmeticko-logické operácie, posun (shift)
6	vetvenie (branch)
7	výpočet adresy pre uloženie dát

3.1.2 Staršie mikroarchitektúry

Keďže predchádzajúcich mikroarchitektúr od firmy Intel je mnoho, zameranie tejto podkapitoly je na mikroarchitektúry, ktoré sú v implementácii vektorových SIMD jednotiek dôležité. Konkrétne sa jedná o mikroarchitektúry

- Sandy Bridge a mikroprocesor s názvom Intel 2nd generation Core,
- P6 (6. generácia Intel x86 mikroarchitektúry) a mikroprocesor s názvom Intel Pentium III.

Sandy Bridge

Sandy Bridge ako prvá mikroarchitektúra od firmy Intel implementovala SIMD technológiu AVX. Počet portov, počet výpočtových jednotiek a ich implementácia v mikroprocesore sú podobné ako u aktuálnej mikroarchitektúry, avšak nie rovnaké. Nasledujúci zoznam uvádza zmeny oproti mikroarchitektúre Haswell [7] [8]:

- Sandy Bridge nepodporuje technológie AVX2 a FMA3, a teda nemá ani výpočtovú jednotku FMA a podporuje len operácie inštrukčnej sady AVX.
- Rezervačná stanica plánuje vykonanie 54och mikroinštrukcií.
- Počet portov pripojených k rezervačnej stanici je celkom 6 (Port 0 až Port 5). Tým pádom Sandy Bridge nemá jednotku AGU a taktiež má o jednu jednotku cALU64 menej.
- Operácie vetvenia, ktoré v mikroarchitektúre Haswell poskytujú porty 0 a 6, sú len na porte 5.

Ostatné operácie, uvedené v tab. 3.1 ostali bez zmeny, takže aj porty s jednotkami uAGU a jednotkou uloženia dát, pracujú rovnako.

P6 (6. generácia Intel x86 mikroarchitektúry)

Mikroprocesor Intel Pentium III v 6. generácii Intel x86 mikroarchitektúry ako prvý implementoval technológiu SSE. Výpočtové porty sú opäť pripojené k rezervačnej stanici pomocou zásobníkov. Rezervačná stanica plánuje vykonanie 20tich mikroinštrukcií. Intel Pentium III má k dispozícii 5 portov označených Port 0 až Port 4 [5]:

- Port 0 - výpočtové jednotky cALU, FPU pre sčítanie (Fadd), SSE a MMX.
- Port 1 - výpočtové jednotky sALU, JEU, FPU pre násobenie (Fmul), SSE a MMX.
- Port 2 - jednotka AGU pre načítanie adresy.
- Port 3 - jednotka AGU pre uloženie adresy.
- Port 4 - jednotka uloženia dát.

Mikroprocesor Intel Pentium III používa len inštrukčný súbor SSE, takže nepodporuje žiadne novšie technológie. Preto neobsahuje výpočtové jednotky AVX a FMA.

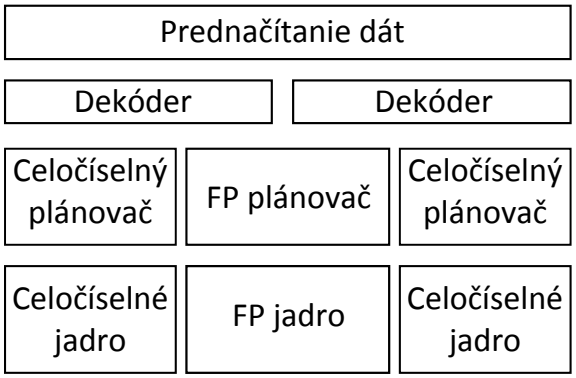
3.2 Firma AMD

3.2.1 Mikroarchitektúra Steamroller

Mikroarchitektúru Steamroller uviedla firma AMD na trh v roku 2014 a v súčasnosti je to najnovšia mikroarchitektúra od firmy AMD. Aj napriek tomu, že AMD mikroarchitektúra Steamroller vyšla až po uvedení Intel mikroarchitektúry Haswell, ktorá používa technológiu AVX2, Steamroller stále využíva predošlú technológiu AVX. Podporuje však inštrukcie sád FMA3 a FMA4.

Dizajn mikroarchitektúry nesie označenie CMT (Clustered Multi-Thread). Táto mikroarchitektúra poskytuje 4 moduly, ktoré sa podľa firmy AMD rovnajú 8 jadrám

(1 modul = 2 jadrá). Toto však nie je pravda. Použitie termínu modul namiesto termínu jadro je vhodnejší a menej mätúci. Jeden modul sa vzhľadom k operačnému systému síce javí ako 2 fyzické jadrá, avšak modul v skutočnosti dve polnohodnotné jadrá neobsahuje. Ide o logické jadrá. Nasledujúci obrázok naznačuje vysvetlenie pojmu modul (obr. 3.2).



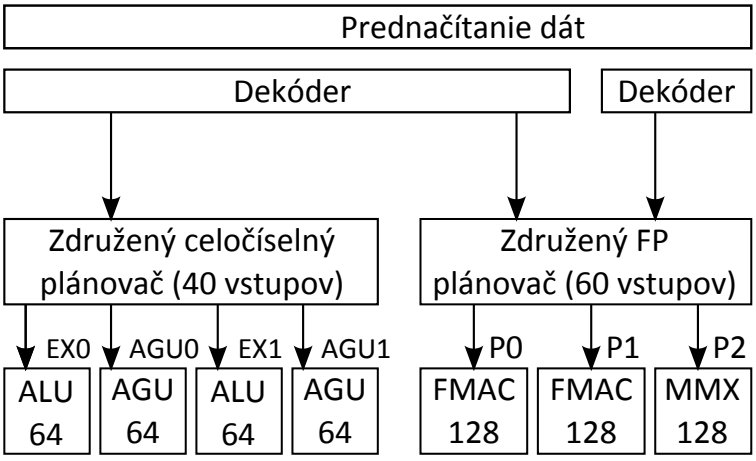
Obr. 3.2: Steamroller - zjednodušená schéma súčastí modulu

Každý dekóder je zdieľaný pre jedno celočíselné jadro a pre FP jadro. Dekódery dokážu pracovať paralelne, čo umožňuje dekódovať 4 inštrukcie v každom cykle pri behu dvoch vlákien zároveň. Celočíselné plánovače sú zdieľané. Oba tieto plánovače majú veľkosť 40 položiek. Plánovač vektorových operácií a operácií s reálnymi číslami s plávajúcou rádovou čiarkou (FP plánovač) má veľkosť 60 položiek. Každý celočíselný plánovač má k dispozícii dve celočíselné 64 bitové aritmeticko-logické jednotky ALU a dve 64 bitové univerzálne jednotky pre uloženie/načítanie adries AGU. Spoločné pre obe jadrá sú jednotky pre ostatné operácie. Pre celočíselné vektorové 128 bitové operácie je k dispozícii jednotka označená ako MMX128 a pre ostatné vektorové výpočty a výpočty reálnych čísiel s plávajúcou rádovou čiarkou sú k dispozícii dve jednotky FMAC128 (Fused-Multiply Accumulate) [5].

Nekompletná schéma modulu mikroarchitektúry Steamroller je na obrázku 3.3. V tejto schéme chýba druhé celočíselné jadro kvôli priestorovému obmedzeniu. Toto druhé celočíselné jadro je však totožné s uvedeným jadrom.

Výpočtové jednotky v celočíselných plánovačoch sú pripojené na výpočtové porty označené EX0, EX1, AGU0 a AGU1. Výpočtové jednotky v plánovači pre FP čísla sú pripojené na výpočtové porty označené P0 - P2. Využitie výpočtových jednotiek na konkrétne operácie popisuje tabuľka 3.2 [6].

Všetky jednotky na portoch P0-P2 majú šírku 128 bitov. V jednom cykle je možné spracovať dve 128 bitové operácie alebo jednu 256 bitovú operáciu. Z toho vyplýva, že pri výpočte jednej AVX operácie musia byť použité obe jednotky FMAC128



Obr. 3.3: Steamroller - skrátená schéma modulu bez druhého celočíselného jadra

Tab. 3.2: Steamroller - operácie na jednotlivých výpočtových portoch

Výpočtový port	**Operácie**
P0	FP násobenie, sčítanie, delenie, logické op., celočíselné vektorové násobenie
P1	FP násobenie, sčítanie, delenie, operácie pomiešania, posunu, zabalenia
P2	celočíselné vektorové sčítanie, vektorové logické op., uloženie dát
EX0	aritmeticko-logické op., delenie
EX1	aritmeticko-logické op., násobenie, skokové operácie
AGU0, AGU1	načítanie/uloženie adresy

súčasne. Problémom však je, že operácia ukladania dát je len na jednej z výpočtových jednotiek, a preto sú nevyhnutné 2 cykly pre uloženie dát so šírkou 256 bitov. Ďalším problémom je to, že táto jednotka ukladania je na rovnakom výpočtovom porte, ktorý používajú 256 bitové AVX inštrukcie. To znamená, že v jednom cykle spracujú súčasne pracujúce jednotky FMAC128 jednu AVX inštrukciu veľkosti 256 bitov a pre jej uloženie sú potrebné ďalšie dva cykly [6]. Mikroprocesory od Intelu nielenže dokážu uložiť 256 bitové dáta v jednom cykle, ale sú schopné spracovať až 3 AVX inštrukcie súčasne.

V mikroarchitektúre Steamroller má každý modul k dispozícii inštrukčnú vyrovnávaciu pamäť Level 1 (L1 cache) veľkosti 96 KB. Pre porovnanie, Intel mikroarchitektúry Sandy Bridge aj Haswell majú veľkosť tejto vyrovnávacej pamäte 32KB

na každé jadro. Veľkosť vyrovnávacej pamäti L1 je teda vyššia u firmy AMD, aj keď je spoločná pre celý modul. Oneskorenie týchto pamätí je však vyššie ako u firmy Intel. Problémom u AMD je neschopnosť znížiť práve toto oneskorenie, a preto sa rozhodli vyskúšať L1 zväčšiť. Všeobecne však platí, že menšia vyrovnávacia pamäť s nižším oneskorením je lepšia ako väčšia vyrovnávacia pamäť s vyšším oneskorením [6] [9].

3.2.2 Staršie mikroarchitektúry

Piledriver a Bulldozer

Mikroarchitektúra Bulldozer ako prvá od firmy AMD poskytovala technológiu AVX. Nadväzujúce mikroarchitektúry Piledriver a Steamroller sú iterácie mikroarchitektúry Bulldozer a všetky používajú dizajn CMT. Mikroarchitektúra Piledriver, ako prvá od AMD, pridáva k inštrukčnej sade FMA4 jej trojoperandovú verziu FMA3.

Mikroarchitektúry Piledriver a Bulldozer sa odlišujú od súčasnej mikroarchitektúry Steamroller v tom, že [7] [10]

- poskytujú o jednu MMX128 jednotku viac. Tá je pripojená na port P3. Odstránenie tejto jednotky v mikroarchitektúre Steamroller sa odôvodňovalo tým, že to má viesť k zachovaniu priepustnosti pri menšej spotrebe energie,
- Steamroller má 2 dekódery, obe prechádzajúce mikroarchitektúry mali len jeden dekóder, ktorý bol zdieľaný pre všetky plánovače v moduli,
- veľkosť vyrovnávacích pamätí pre inštrukcie L1 je 64 KB, nie 96 KB.

4 INŠTRUKČNÉ SADY

4.1 Inštrukčná sada SSE

Inštrukčná sada SSE poskytuje súbor inštrukcií, ktoré sú rozdelené do štyroch hlavných skupín [2]:

1. Skalárne a vektorové inštrukcie pre prácu s reálnymi číslami s jednoduchou presnosťou.
2. 64bitové SIMD inštrukcie pre prácu s celými číslami.
3. Inštrukcie pre správu stavu jednotky.
4. Inštrukcie pre kontrolu vyrovnávacej pamäte a usporiadania pamäte.

Skalárne a vektorové inštrukcie sa ďalej delia na

- operácie posunu,
- logické operácie,
- aritmetické operácie,
- porovnávacie operácie,
- operácie pre konverziu medzi číselnými typmi,
- operácie pomiešania.

Zoznam inštrukcií vrátane ich popisu a správania poskytuje samostatná príloha.

4.2 Inštrukčná sada SSE2

Inštrukčný súbor SSE2 pridáva k predošlým inštrukčným sadám inštrukcie, ktoré je možné rozdeliť do štyroch skupín [2]:

1. Inštrukcie podporujúce vektorové a skalárne operácie s reálnymi číslami s dvojitou presnosťou.
2. Celočíselné 64bitové a 128bitové inštrukcie.
3. 128bitové rozšírenie celočíselných inštrukcií poskytnutých v rámci predošlých inštrukčných súborov.
4. Inštrukcie pre zoradenie inštrukcií v kóde a pre kontrolu vyrovnávacej pamäte.

Inštrukcie pre prácu s reálnymi číslami s dvojitou presnosťou sa ďalej delia na

- operácie pre pohyb dát,
- aritmetické operácie,
- porovnávacie operácie,
- operácie pre konverziu,
- logické operácie,
- operácie pomiešania.

Zoznam inštrukcií vrátane ich popisu a správania poskytuje samostatná príloha.

4.3 Inštrukčná sada SSE3

Inštrukčná sada SSE3 pridáva k predošlým inštrukciám 13 nových inštrukcií. Konkrétne [2]

- 1 inštrukciu, ktorá vylepšuje konverziu reálneho čísla na celé číslo,
- 1 inštrukciu, ktorá poskytuje načítanie 128 bitových nezarovnaných dát,
- 3 inštrukcie, ktoré vylepšujú výkon pri operáciách načítania, pohybu a duplikovania,
- 2 inštrukcie, ktoré poskytujú vektorové sčítanie a odčítanie,
- 4 inštrukcie, ktoré poskytujú horizontálne sčítanie a odčítanie,
- 2 inštrukcie, ktoré zlepšujú synchronizáciu medzi viacvláknovými prvkami.

Zoznam inštrukcií vrátane ich popisu a správania poskytuje samostatná príloha.

4.4 Inštrukčná sada SSSE3

Súbor inštrukcií SSSE3 obsahuje 32 nových inštrukcií. Je to [2]

- 12 inštrukcií, ktoré vykonávajú operácie horizontálneho sčítania alebo odčítania,
- 6 inštrukcií, ktoré vyčíslujú absolútne hodnoty,
- 2 inštrukcie, ktoré vykonávajú operácie násobenia a sčítania a zrýchľujú vyčíslovanie desatinných čiarok,
- 2 inštrukcie, ktoré urýchľujú vektorové celočíselné násobenie,
- 2 inštrukcie, ktoré vylepšujú operácie pomiešania,
- 6 inštrukcií, ktoré negujú vektorové celé čísla v cieľovom operande, ak znaky odpovedajúceho prvku v zdrojovom operande sú menšie ako nula,
- 2 inštrukcie, ktoré zarovnajú dáta zložené z dvoch operandov.

Zoznam inštrukcií vrátane ich popisu a správania poskytuje samostatná príloha.

4.5 Inštrukčná sada SSE4.1

Inštrukčná sada SSE4.1 pridáva 47 nových inštrukcií, ktoré sú zamerané na zlepšenie výkonu pri práci s 3D grafikou a multimédiami [2]. Ich kompletný zoznam spolu s popisom a správaním jednotlivých inštrukcií poskytuje samostatná príloha k práci.

4.6 Inštrukčná sada SSE4.2

Celkovo 19 nových inštrukcií bolo pridaných v rámci SSE4.2 k predošlým inštrukčným sadám. Tie vylepšujú výkon [2]

- v oblasti znakového a textového spracovania, ktoré využíva SIMD techniku,
- v oblasti celočíselných SIMD inštrukcií, ktoré zlepšujú schopnosti práce s celočíselnými 128 bitovými SIMD operáciami použitými v SSE4.1.

Zoznam inštrukcií vrátane ich popisu a správania poskytuje samostatná príloha.

4.7 Inštrukčná sada AVX

Inštrukčný súbor AVX prináša mnoho nových inštrukcií. Väčšina z nich poskytuje oproti predošlým inštrukciám ich ekvivalenty, ktoré sú kódované schémou VEX, a teda podporujú nedeštruktívny operand. Inštrukčná sada AVX ako prvá poskytuje inštrukcie pre prácu s dátami šírky 256 bitov [2]. Zoznam inštrukcií spolu s ich popisom a správaním poskytuje samostatná príloha.

4.8 Inštrukčná sada AVX2

Súbor inštrukcií AVX2 prináša oproti AVX mnoho 256bitových celočíselných SIMD inštrukcií, ktoré predtým pracovali ako 128bitové. Okrem toho AVX2 poskytuje vylepšenú funkčnosť permutačných operácií na jednotlivých dátových prvkoch, vylepšenú funkčnosť vektorových inštrukcií posunu s variabilným počtom posunov na dátový prvok a prináša tiež inštrukcie pre načítanie nesúvislých dátových prvkov z pamäte. AVX2 inštrukcie majú rovnaký programovací model ako AVX. Sú kódované použitím schémy VEX [2]. Zoznam inštrukcií vrátane ich popisu a správania poskytuje samostatná príloha.

4.9 Inštrukčná sada FMA3

Inštrukčná sada FMA3 pridáva nové inštrukcie, ktoré umožňujú vykonať dve matematické operácie v jednom kroku, čím zvyšuje výkon a priepustnosť. Zoznam týchto inštrukcií vrátane ich popisu a správania poskytuje samostatná príloha.

5 INŠTRUKCIA CPUID

Inštrukcia CPUID (CPU Identification) spočiatku poskytovala informácie o procesore. Postupom času a vývojom procesorov bola táto inštrukcia vyvinutá tak, aby postkytovala nielen informácie o samotnom procesore, ale aj o podpore konkrétnych funkcií a vlastností v rámci daného procesora. Predpokladá sa, že evolúcia procesorov bude pokračovať, a preto je inštrukcia CPUID navrhnutá tak, aby bola rozšíriteľná [11].

5.1 Výstup CPUID inštrukcie

CPUID inštrukcia podporuje dve sady funkcií. Jedna poskytuje jednoduché informácie o procesore a druhá rozšírené informácie o procesore. Výstup je plne závislý od obsahu registra EAX (Extended Accumulator). To znamená, že po spustení inštrukcie CPUID, táto vykoná špecifickú funkciu závislú na hodnote registra EAX.

Na zistenie najvyššej možnej hodnoty, ktorá môže byť vložená do registra EAX a vráti základné informácie o procesore, je možné využiť hodnotu 0 v tomto registri a následné spustenie CPUID inštrukcie.

```
MOV EAX, 00h
CPUID
```

Návratová hodnota bude po tomto kroku dostupná z registra EAX. Pri zisťovaní podpory rôznych vlastností procesoru by teda hodnota registra EAX mala byť vyššia ako 0 a nižšia alebo rovná tejto návratovej hodnote.

Zistenie najvyššej možnej hodnoty, ktorá môže byť vložená do registra EAX a vracia rožšírené informácie o procesore, je možné využiť hodnotu 80000000h a následné spustenie CPUID inštrukcie.

```
MOV EAX, 80000000h
CPUID
```

Návratová hodnota je opäť dostupná z registra EAX. Pri zisťovaní podpory rôznych vlastností procesoru by teda hodnota registra EAX mala byť vyššia ako 80000000h a nižšia alebo rovná tejto návratovej hodnote.

Na terajších a budúcich procesoroch Intel achitektúry IA-32 je bit 31 v registri EAX vynulovaný v prípade, že je inštrukcia CPUID spustená s parametrom vyšším ako je najvyššia možná hodnota a ak procesor nepodporuje danú funkciu [11].

5.2 Značky vlastností

Pri zadaní hodnoty do registra EAX, prípadne spolu so zadaním hodnoty do ECX a spustení inštrukcie CPUID, táto načíta dátový register EDX (Extended Data) a počítadlový register ECX (Extended Counter), prípadne bázový register EBX (Extended Base) so značkami daných vlastností. Jednotlivé bity registrov indikujú, ktorú vlastnosť procesor podporuje [11]. Niektoré návratové bity indikujú podporu technológie, ktorá zatiaľ nie je dostupná na trhu a má byť rozšírením technológie AVX2. Táto pripravovaná technológia sa nazýva AVX-512.

EAX=1

Pokiaľ je register EAX nastavený na hodnotu 1, jednotlivé bity registrov EDX a ECX po spustení CPUID inštrukcie indikujú vlastnosti podporované v rámci procesora, ako uvádzajú tabuľky 5.1 a 5.2.

Tab. 5.1: Značky vlastnotí jednotlivých bitov registra EDX pri hodnote EAX=1

Bit	**Vlastnosť**	**Bit**	**Vlastnosť**
0	Podpora x87 FPU	7	Podpora kontroly zariadenia MCE (Machine Check Exception).
1	Virtuálny mód rozšírení 8086	8	Podpora CMPXCHG8 inštrukcie (compare-and-swap)
2	Podpora rozšírenia ladenia	9	Podpora APIC (Advanced Programmable Interrupt Controller)
3	Rozšírenie veľkostí strán	10	Rezervované
4	Podpora registra TSC (Time Stamp Counter)	11	Podpora inštrukcií SYSENTER a SYSEXIT
5	Podpora MSR registrov (Model-Specific Register)	12	Podpora registrov MTRR (Memory Type Range Registers)
6	Podpora rozšírenia fyzickej adresy	13	Podpora PGE (Page Global Enable) bitu v CR4 (Control Register 4)

Bit	Vlastnosť	Bit	Vlastnosť
14	Podpora MCA (Machine Check Architecture)	23	Podpora MMX rozšírenia
15	Podpora inštrukcií podmieneného pohybu a FCMOV inštrukcií	24	Podpora inštrukcií FXSAVE, FXRESTORE
16	Podpora PAT (Page Attribute Table)	25	Podpora SSE rozšírenia
17	Podpora rozšírenia 36bitovej veľkosti strany	26	Podpora SSE2 rozšírenia
18	Sériové číslo procesoru	27	Vyrovnávacia pamäť CPU podporuje samopozorovanie
19	Podpora inštrukcie CLFLUSH	28	Podpora viacvláknových operácií
20	Rezervované	29	Podpora automatického obmedzenia teploty popocou termálneho sledovača
21	Podpora ukladania ladenia	30	Emulovanie x86 (ak je procesor architektúry IA-64)
22	Podpora termálnej kontroly MSR registrov	31	Podpora PBE (Pending Break Enable)

Tab. 5.2: Značky vlastností jednotlivých bitov registra ECX pri hodnote EAX=1

Bit	Vlastnosť	Bit	Vlastnosť
0	Podpora SSE3 rozšírenia	1	Podpora inštrukcie PCLMULQDQ

Bit	Vlastnosť	Bit	Vlastnosť
2	Ukladanie 64bi-tového ladenia	17	Identifikácia kontextu procesov
3	Podpora inštrukcií MONITOR a MWAIT	18	Podpora priameho prístupu do vyrovnávacej pamäte pre priame zápisy do pamätí
4	Ukladanie kvalifoko-vaného ladenia CPL (Control Panel Aplet)	19	Podpora rozšírenia SSE4.1
5	Podpora rozšírení VMX (Virtual Machine Ext.)	20	Podpora rozšírenia SSE4.2
6	Rozšírenia SMX (Safer Mode Extenisions)	21	Podpora x2APIC
7	Podpora vylepšenej funkcie SpeedStep	22	Podpora inštrukcie MOVBE
8	Podpora termálneho sledovača 2 (TM2)	23	Podpora inštrukcie POPCNT
9	Podpora SSSE3 rozší-renia	24	Podpora jednorázovej APIC operácie s použitím konеč-nej hodnoty TSC
10	Identifikátor kontextu vyrovnávacej pamäte L1	25	Podpora AES rozšírenia
11	Podpora rozhrania Silicon Debug	26	Podpora inštrukcií XSAVE XSTOR, XSETBV, XGETBV
12	Podpora FMA3 rozšírenia	27	Zistenie povolenia inštrukcie XSAVE operačným systémom
13	Podpora inštrukcie CMPXCHG16B	28	Podpora AVX rozšírenia
14	Vypnutie zasielania správ o priorite úloh	29	Podpora polovičnej presnosti F16C (half-precision)
15	Podpora Perfmon	30	Podpora RDRAND (Random Number Generator)
16	Rezervované	31	Podpora sledovača virtuálneho zariadenia (tzv. Hypervisor)

EAX=7, ECX=0

Po nastavení hodnôt EAX=7 a ECX=0 a spustení inštrukcie CPUID, táto načíta značky rozšírených vlastností procesora. Jednotlivé návratové hodnoty bitov registra EBX indikujú rozšírené vlastnosti podporované v rámci procesora, ako uvádza tabuľka 5.3.

Pri spustení inštrukcie CPUID a hodnotách EAX=7, ECX=0, sú návratové hodnoty bitov 2 až 31 registra ECX rezervované. Bit 0 indikuje podporu inštrukcie PREFETCHWT1 a bit 2 podporu AVX-512 inštrukcií pre manipuláciu s bitmi.

Tab. 5.3: Značky vlastnotí jednotlivých bitov registra EBX pri hodnote EAX=7, ECX=0

Bit	**Vlastnosť**	**Bit**	**Vlastnosť**
0	Prístup k základni %fs a %gs	13	Rezervované
1	Rezervované	14	Podpora Intel MPX rozšírenia (Memory Protection Extens.)
2	Rezervované	15	Rezervované
3	Podpora rozšírenia BMI1 (Bit Manipulation Instruction)	16	Podpora rozšírenia AVX-512
4	Podpora rozšírení TSX (Transactional Synchronization Extensions)	17	Podpora AVX-512 doubleword a quadword inštrukcií
5	Podpora AVX2 rozšírenia	18	Podpora inštrukcie RDSEED
6	Rezervované	19	Podpora Intel ADX (Multi-Precision Add-Carry Instruction Extensions)
7	Podpora SMEP (Supervisor Mode Execution Prevention)	20	Podpora SMAP (Supervisor Mode Access Prevention)
8	Podpora rozšírenia BMI2	21	Podpora AVX-512 inštrukcií pre celočíselné FMA operácie
9	Podpora inštrukcií REP MOVSB/STOSB	22	Podpora PCOMMIT inštrukcie
10	Podpora inštrukcie INVPCID	23	Podpora CLFLUCHOPT inštrukcie
11	Podpora rozšírení TSX	24	Podpora CLWB inštrukcie
12	Rezervované	25	Podpora IPT (Intel Processor Trace)

Bit	Vlastnosť	Bit	Vlastnosť
26	Podpora AVX-512 inštrukcií prednačítania (prefetch instructions)	29	Podpora Intel SHA (Secure Hash Algorithm)
27	Podpora AVX-512 exponenciálnych inštrukcií	30	Podpora AVX-512 byte a word inštrukcií
28	Podpora AVX-512 inštrukcií pre zistenie konfliktov	31	Podpora rozšírení AVX-512 VLE (Vector Lenght Extens.)

6 OPERÁCIE S MATICAMI A VEKTORMI

Vďaka veľkým nárokom stále rýchlejšie sa rozvíjajúcich počítačových hier a 3D grafiky vôbec, vznikli tzv. vektorové inštrukcie pre prácu s viacerými hodnotami v jednom kroku. Vznikli najmä preto, lebo v týchto oblastiach hrajú veľkú rolu maticové a vektorové výpočty. Inštrukcie pracujúce s vektormi dát výrazne urýchľujú tieto výpočty. Nasledujúce kapitoly sa preto budú zaoberať realizáciou elementárnych operácií s vektormi a maticami.

6.1 Realizácia základných výpočtov pomocou vektorových výpočtových jednotiek

Pri využívaní vektorových výpočtových jednotiek je nutné uvedomiť si, akým spôsobom procesor spracováva jednotlivé inštrukcie, s akými dátovými typmi pracujeme a ako sa budú jednotlivé prvky v registroch meniť (viď. kap. 2). V nasledujúcich kapitolách budú popísané operácie s vektormi a maticami, predovšetkým ich transpozícia a násobenie vektora s maticou (tzv. násobenie zľava), resp. násobenie matice vektorom (tzv. násobenie sprava) a tiež jednotlivé inštrukcie použité v ukážkach zdrojových kódov. Pre jednoduchosť budeme uvažovať maticu rozmerov 4x4 a vektor 1x4, resp. 4x1.

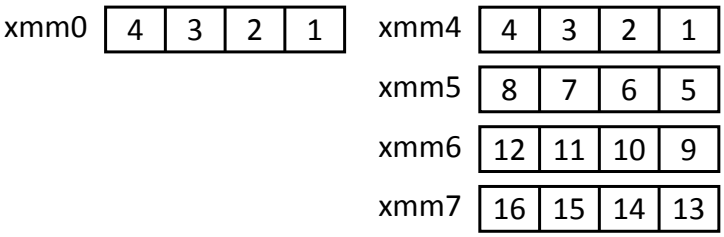
6.1.1 Transpozícia vektora

Transpozícia vektora je nutná pri násobení vektora maticou, tj. pri násobení zľava (kap. 6.1.2). Pri tomto type násobenia sa riadkovo orientovaný vektor násobí s každým stĺpcom matice a následne sa tieto hodnoty sčítajú do daného prvku výsledného vektora (obr. 6.1). V prípade výpočtu pomocou vektorovej výpočtovej jednotky je

$$\begin{pmatrix}1 & 2 & 3 & 4\end{pmatrix} \times \begin{pmatrix}1 & 2 & 3 & 4\\ 5 & 6 & 7 & 8\\ 9 & 10 & 11 & 12\\ 13 & 14 & 15 & 16\end{pmatrix} = \begin{pmatrix}90 & 100 & 110 & 120\end{pmatrix}$$

Obr. 6.1: Algebraické znázornenie výpočtu vektora násobeného maticou

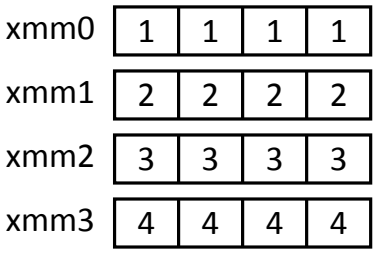
dôležité uvedomiť si, ako bude vektor a matica uložená v daných registroch. Prvky budú ukladané opačne, pretože sa načítavajú do registrov od najnižších adries, ktoré



Obr. 6.2: Spôsob uloženia hodnôt vektora a matice do XMM registrov

sú vpravo (obr. 2.1). Ďalej je potrebné si uvedomiť, akým spôsob realizuje výpočet človek a ako ho bude realizovať procesor. Zásadnou vecou je fakt, že inštrukcie SSE podporujú dvojoperandový formát (kap. 2.1.3) a inštrukcie AVX a FMA3 trojoperandový formát (kap. 2.2.3). To znamená, že pri výpočte nie sme schopní vykonať krok z obrázku 6.1, tj. vynásobenie prvého stĺpca matice vektorom, pretože prvý stĺpec matice je uložený až v štyroch XMM registroch (obr. 6.2). V kapitole 2.1.3 je na obrázku 2.3 znázornené vektorové sčítanie dvoch XMM registrov. V podstate pri všetkých vektorových operáciách platí, že XMM registre pracujú „pod sebou“.

Jedným z riešení tohto problému je transpozícia vektora. Využijeme fakt, že prvá hodnota vektora, teda číslo 1, sa bude násobiť s každou hodnotou v prvom riadku matice. Pomocou dvoch typov inštrukcií, popísaných v kapitole 7.1, dokážeme konkrétnu hodnotu vektora, ktorou budeme násobiť konkrétny riadok matice, uložiť do jedného XMM registra. Po sekvencií daných inštrukcií bude vektor transponovaný a uložený v XMM registroch tak, ako ilustruje nasledujúci obrázok (obr. 6.3).

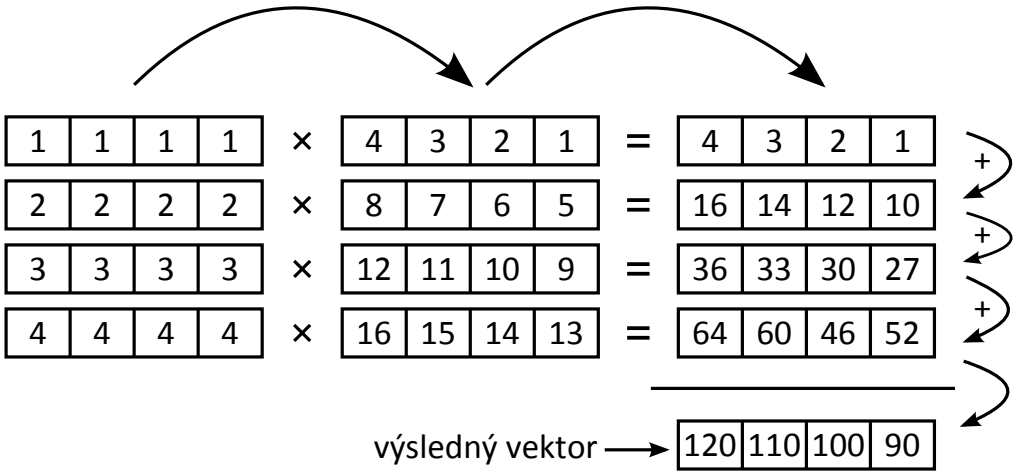


Obr. 6.3: Obsah XMM registrov po transpozícii vektora

Optimalizovaný zdrojový kód transpozície vektora v oboch variantách poskytuje kapitola 7.3.1.

6.1.2 Násobenie vektora maticou

Pri tzv. násobení zľava využijeme predošlé poznatky a transpozíciu vektora. Vhodne zvolenou sekvenciou inštrukcií vynásobíme vektor maticou a dostaneme výsledný vektor. Priebeh tohto procesu ilustruje nasledujúci obrázok (obr. 6.4). Po vynásobení

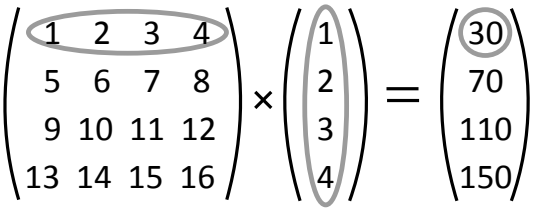


Obr. 6.4: Postup násobenia matice vektorom s hodnotami registrov XMM

vstupných hodnôt dostávame medzivýsledok uložený v štyroch XMM registroch. Tento sa vektorovo sčíta a výsledkom je výstupný vektor uložený v jednom XMM registri.

6.1.3 Násobenie matice vektorom

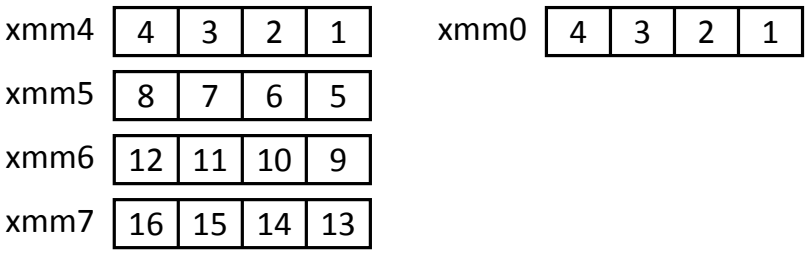
Druhým typom násobenia vektorov a matíc je tzv. násobenie sprava, kedy sa každý riadok matice násobí so stĺpcovo orientovaným vektorom a výsledkom je opäť stĺpcovo orientovaný vektor. (obr. 6.5). Tento typ násobenia je z pohľadu výpočtovej



Obr. 6.5: Algebraické znázornenie výpočtu matice násobenej vektorom

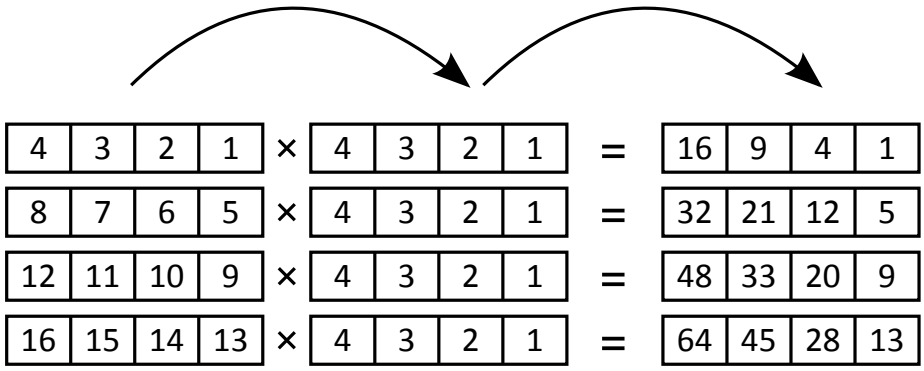
jednotky, od verzie inštrukčnej sady SSE3, výhodnejší ako násobenie zľava. Najväčšia výhoda je v tom, že vstupný vektor nemusíme transponovať, pretože jeho samotné

uloženie do XMM registra je zároveň jeho transpozíciou. Vstupné hodnoty v XMM registroch budú vyzerať nasledovne (obr. 6.6). Keďže vstupný vektor nemusíme



Obr. 6.6: Uložené vstupné hodnoty matice a vektora v XMM registroch

transponovať, sekvencia inštrukcií pre výpočet výstupného vektora sa skráti. Hodnoty riadkov matice vynásobíme priamo so vstupným vektorom (obr. 6.7). Následne

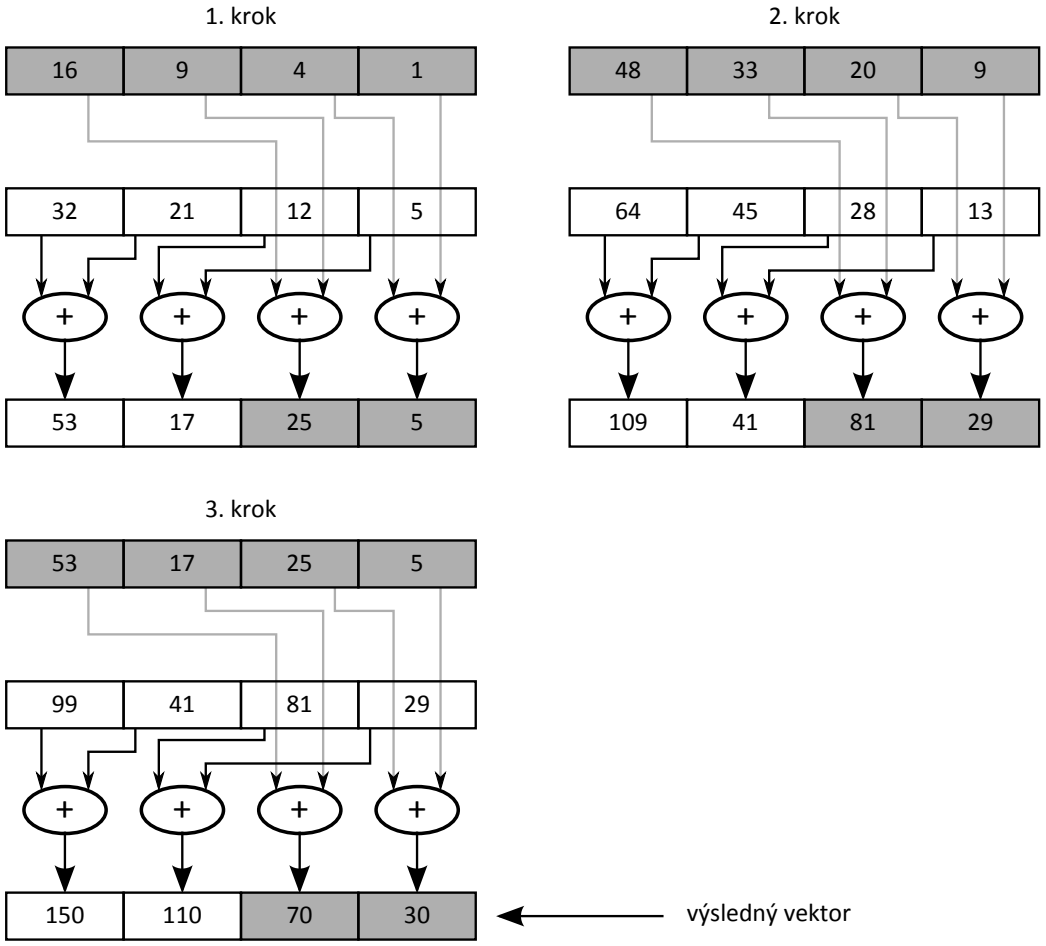


Obr. 6.7: Uložené vstupné hodnoty matice a vektora v XMM registroch

však už nemôžeme uplatniť klasické sčítanie XMM registrov, pretože hodnoty, ktoré potrebujeme sčítať sa nenachádzajú „pod sebou“, ale v každom XMM registri zvlášť. Preto využijeme špeciálny typ sčítania pomocou inštrukcie haddps (Packed Single-FP Horizontal Add). Postup sčítania ilustruje obrázok 6.8. Z obrázku je zrejmé, ako inštrukcia haddps funguje, a preto sa jej popis nenachádza v ďalších kapitolách, kde sú vysvetlené ostatné využité inštrukcie.

6.1.4 Transpozícia matice

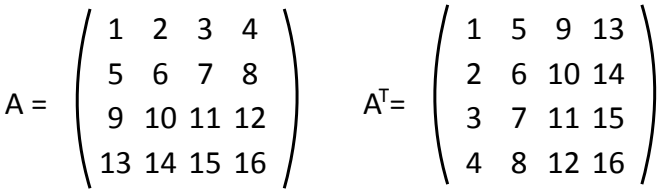
Transpozícia matice je výpočtovo náročná operácia, preto je vhodné sa jej vyhnúť v čo najviac prípadoch. Ak však nie je iné riešenie, je pri SSE možné transponovať maticu s využitím skupiny tzv. rozbalovacích inštrukcií (unpack). Od inštrukčnej



Obr. 6.8: Postup sčítavania pomocou inštrukcie haddps

sady AVX sú k dispozícii permutačné inštrukcie, vďaka ktorým je, teoreticky, transpozícia o niečo rýchlejšia, avšak stále platí, že je to výpočtovo náročný proces.

Pri transpozícii matice sa jednotlivé riadky matice vymenia so stĺpcami (obr. 6.9). Pri použití inštrukcií SSE je na transpozíciu matice potrebný sled 12tich inštrukcií. Pri AVX sa tento výpočet skráti o 4 inštrukcie. Počet XMM, resp. YMM registrov potrebných na transpozíciu je pri oboch výpočtových jednotkách 5. Praktické využitie transpozície matice ukazujú zdrojové kódy v kapitole 7.3.4.



Obr. 6.9: Algebraické znázornenie transpozície matice 4x4

7 PRAKTICKÁ REALIZÁCIA VÝPOČTOV

Nasledujúce kapitoly poskytujú ukážky zdrojových kódov a popis inštrukcií využitých pri realizácii základných výpočtov s maticami a vektormi. Popis týchto inštrukcií je tiež možné nájsť v samostatnej prílohe k práci, avšak tá neobsahuje grafické znázornenie správania inštrukcií, a preto si v tejto kapitole priblížime detailnejšie inštrukcie použité v zdrojových kódoch v nasledujúcich kapitolách. Inštrukcie sú rozdelené do troch častí.

- Inštrukcie pri využití výpočtovej jednotky SSE.
- Inštrukcie pri využití výpočtovej jednotky AVX.
- Inštrukcie pri využití výpočtovej jednotky FMA3.

Popis ostatných inštrukcií poskytuje samostatná príloha k práci.

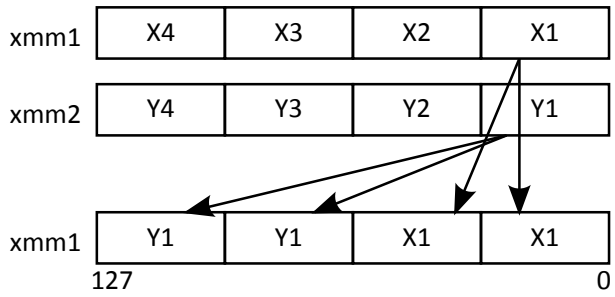
7.1 Inštruckie SSE

Inštruckia movups

Movups (Move Unaligned Sigle-Precision FP Values) je základná inštrukcia naplnenia. Pri využití tejto inštrukcie sa hodnoty zo zdrojového operandu skopírujú do cieľového operandu, takže dostaneme dva úplne totožné XMM registre.

Inštrukcia shufps

Inštrukciu shufps (Shuffle Packed Single-Precision Floating-Point Values) budeme využívať pri transpozícii vektora. Shufps inštrukcia uloží dve zo štyroch hodnôt prvého registra do spodnej polovice cieľového registra. Rovnako uloží dve zo štyroch hodnôt druhého registra, ale do hornej polovice cieľového registra. To, ktoré hodnoty budú vybrané definuje 8bitová maska použitá v syntaxe inštrukcie. Správanie tejto inštrukcie ilustruje obrázok 7.1. Ak použijeme ako zdrojový a cieľový operand

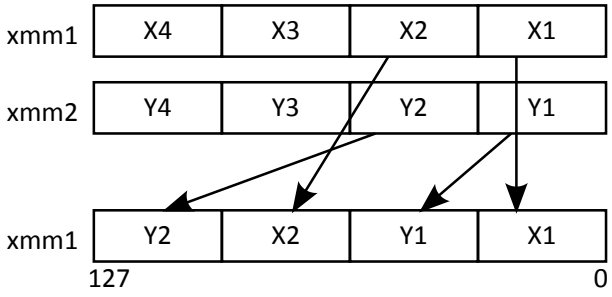


Obr. 7.1: Príklad operácie prevedenej inštrukciou shufps

zhodný XMM register, sme schopní jednou vybranou hodnotou naplniť daný register a sekvenciou týchto inštrukcií transponujeme vektor, ako bolo uvedené v kapitole 6.1.1.

Inštrukcie unpcklps/unpckhps

Rovnaký výsledok ako pri použití inštrukcie shufps dostaneme s využitím inštrukcií unpcklps, resp. unpckhps (Unpack and Interleave Low/High Packed Single-Precision Floating-Point Values). Rozdiel medzi týmito inštrukciami je, ako už názov napovedá, v práci s prvkami buď v dolnej polovici alebo v hornej polovici XMM registra. Podobne ako pri inštrukcii shufps sa vyberú dve zo štyroch hodnôt v jednom registri a uložia sa preložene s hodnotami v druhom registry. Hodnoty sa tentoraz nevyberajú pomocou 8bitovej masky, ale definujú ich priamo inštrukcie (high/low). Nasledujúci obrázok približuje správanie inštrukcie pre výber spodnej časti dát (low) (obr. 7.2). Opäť platí, že ak použijeme ako zdrojový a cieľový operand rovnaký XMM



Obr. 7.2: Príklad operácie prevedenej inštrukciou unpcklps

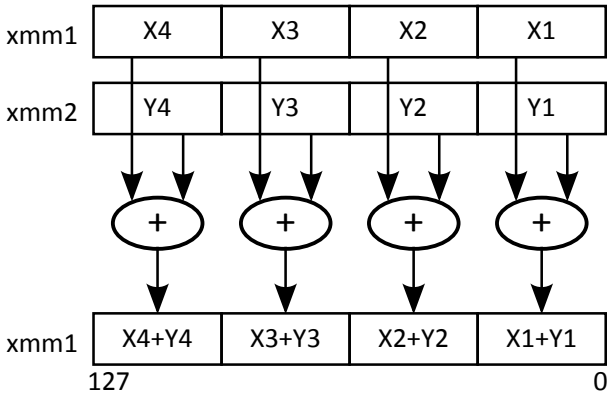
register, správnou sekvenciou príkazov transponujeme vektor.

Inštrukcia addps

Pri použití tejto inštrukcie sa vektorovo sčítajú hodnoty jednotlivých registrov. Správanie tejto inštrukcie je ľahko pochopiteľné a ilustruje ho nasledujúci obrázok (obr. 7.3).

Inštrukcia haddps

Správanie tejto inštrukcie bolo vysvetlené a názorne zobrazené v kapitole 6.1.3 a na obrázku 6.8.



Obr. 7.3: Operácia prevedená inštrukciou addps

Inštrukcia mulps

Inštrukcia mulps (Multiply Packed Single-Precision Floating-Point Values) je veľmi podobná inštruckii addps, avšak, ako je už z názvu zrejmé, táto inštrukcia hodnoty medzi sebou nesčítava, ale násobí.

Inštrukcie rozbalenia

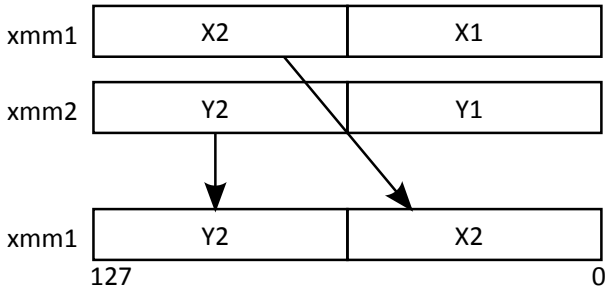
Skupinu inštrukcií rozbalenia (Unpack Data) tvorí niekoľko inštrukcií, z ktorých budeme využívať štyri pri transpozícii matice. Sú to inštrukcie

- pre prácu s 32bitovými prvkami punpckldq a punpckhdq,
- pre prácu so 64bitovými prvkami punpcklqdq a punpckhqdq.

Inštruckie pre prácu s 32bitovými hodnotami (punpck inštrukcie) pracujú rovnako ako inštrukcie unpcklps a unpckhps (viď. obr. 7.2). Tiež je definícia výberu prvkov z registra daná priamo inštrukciou (low/high). Pri použití inštrukcií pre prácu so 64bitovými prvkami sa pracuje buď s dolnou alebo hornou polovicou XMM registra. Správanie inštrukcie punpckhqdq zobrazuje nasledujúci obrázok (obr. 7.4).

7.2 Inštrukcie AVX a FMA3

Pri využití inštrukčných súborov AVX a vyššie sa správanie inštrukcií mení. Niektoré inštrukcie, využité pri výpočtovej jednotke SSE, nahradia ekvivalentné inštrukcie pri použití AVX a FMA3. Jediný rozdiel bude v zápise týchto inštrukcií. Od výpočtovej jednotky AVX sú k diposzícii 2x väčšie registre YMM (kap. 2.2.1), a preto sú inštrukcie kódované pre využitie týchto registrov kódovacou schémou VEX-prefix (kap. 2.2.3).



Obr. 7.4: Príklad operácie prevedenej inštrukciou punpckhps

Len niektoré Vex-prefixové verzie inštrukcií, použité pri SSE, bude možné použiť aj pri AVX a FM3. Nasleduje ich zoznam:

- Inštrukciu movups nahradí **vmovupd** (Vex Move Unaligned Packed Double-Precision FP Values).
- Inštrukciu mulps nahradí **vmulpd** (Vex Multiply Packed Double-Precision FP Values).
- Inštrukciu addps nahradí **vaddpd** (Vex Add Packed Double-Precision FP Values).
- Inštrukcie unpcklps/unpcklpd nahradia **vunpcklpd/vunpckhpd** (Vex Unpack Low/High Packed Double-Precision FP Values).

Problémom však je, že YMM registre sa pri niektorých inštrukciách správajú ako dva zlúčené XMM registre, takže zdrojové kódy použité pri SSE nestačí len upraviť na formát Vex prefixu. Výpočty pomocou AVX a FMA3 vyžadujú použitie nových inštrukcií popísaných v nasledujúcej kapitole.

Inštrukcia vperm2f128

Inštrukcia permutácie skopíruje na základe 8bitovej masky jednu polovicu (0tý až 127mi bit alebo 128mi až 255ty bit) z ktoréhokoľvek zdrojového registra a uloží ju do dolnej polovice cieľového registra. Rovnako postupuje aj pri kopírovaní hodnôt do hornej polovice cieľového registra.

Operácia:
Cieľ[0..127]=Zdroj1[0..127] alebo Zdroj1[128..255] alebo Zdroj2[0..127] alebo Zdroj2[128..255]
Cieľ[128..255]=Zdroj1[0..127] alebo Zdroj1[128..255] alebo Zdroj2[0..127] alebo Zdroj2[128..255]

Inštrukcia vbroadcastsd

Už z názvu inštrukcie je zreteľné, že jej operáciou bude nahranie jednej hodnoty do celého YMM registra. Túto inštrukciu využijeme pri transpozícii vektora, kde z pamäte nahráme jednu hodnotu vstupného vektora do všetkých elementov YMM registra.

Inštrukcia vfmadd231pd

Inštrukcia súboru FMA3. V podstate spája inštrukciu násobenia (kap. 7.1) s inštrukciou sčítania (kap. 7.1) do jedného kroku. Hodnoty zdrojových registrov sú vektorovo vynásobené a celkový výsledok sa vektorovo pričíta k hodnotám v cieľovom registri. Najväčšou výhodou je vyššia presnosť pri zaokrúhľovaní.

7.3 Zdrojové kódy

7.3.1 Transpozícia vektora - SSE

Prvá varianta (inštrukcie shufps):

```
movups xmm0, [esi]    ;esi: ukazateľ na vektor, xmm0: v4, v3, v2, v1
movups xmm1, xmm0     ;xmm1: v4, v3, v2, v1
shufps xmm1, xmm1, 0x00    ;xmm1: v1, v1, v1, v1
movups xmm2, xmm0     ;xmm2: v4, v3, v2, v1
shufps xmm2, xmm2, 0x55    ;xmm2: v2, v2, v2, v2
movups xmm3, xmm0     ;xmm3: v4, v3, v2, v1
shufps xmm3, xmm3, 0xAA    ;xmm3: v3, v3, v3, v3
movups xmm4, xmm0     ;xmm4: v4, v3, v2, v1
shufps xmm4, xmm4, 0xFF    ;xmm4: v4, v4, v4, v4
```

Druhá varianta (unpcklps/unpckhps):

```
movups xmm0, [esi]    ;esi: ukazateľ na vektor, xmm0: v4, v3, v2, v1
movups xmm1, xmm0     ;xmm1: v4, v3, v2, v1
unpcklps xmm1, xmm1     ;xmm1: v2, v2, v1, v1
movups xmm2, xmm1     ;xmm2: v2, v2, v1, v1
unpcklps xmm1, xmm1     ;xmm1: v1, v1, v1, v1
unpckhps xmm2, xmm2     ;xmm2: v2, v2, v2, v2
movups xmm3, xmm0     ;xmm3: v4, v3, v2, v1
unpckhps xmm3, xmm3     ;xmm3: v4, v4, v3, v3
movups xmm4, xmm3     ;xmm4: v4, v4, v3, v3
```

```
unpcklps xmm3, xmm3    ;xmm3: v3, v3, v3, v3
unpckhps xmm4, xmm4    ;xmm4: v4, v4, v4, v4
```

7.3.2 Transpozícia vektora - AVX

Transpozícia vektora sa s pomocou AVX inštrukcií realizuje odlišným spôsobom, ako pri výpočtovej jednotke SSE. Na tento proces sú využité pri výpočtovej jednotek AVX dve varianty kódov.

- S využitím broadcastových inštrukcií.
- Varianta s využitím permutačných inštrukcií a inštrukcií rozbalenia.

Prvá varianta:

```
;esi = ukazateľ na vektor
vbroadcastsd ymm0, [esi]    ;ymm0: v1, v1, v1, v1
add esi,8    posun ukazateľa
vbroadcastsd ymm1, [esi]    ;ymm1: v2, v2, v2, v2
add esi,8    posun ukazateľa
vbroadcastsd ymm2, [esi]    ;ymm2: v3, v3, v3, v3
add esi,8    posun ukazateľa
vbroadcastsd ymm3, [esi]    ;ymm3: v4, v4 v4, v4
```

Druhá varianta:

```
;esi = ukazateľ na vektor
vmovupd ymm7,[esi]    ;ymm7: v4, v3, v2, v1
vunpcklpd ymm0,ymm7,ymm7
vperm2f128 ymm0,ymm0,ymm0,0    ;ymm0: v1, v1, v1, v1
vunpckhpd ymm1,ymm7,ymm7
vperm2f128 ymm1,ymm1,ymm1,17    ;ymm1: v2, v2, v2, v2
vunpcklpd ymm2,ymm7,ymm7
vperm2f128 ymm2,ymm2,ymm2,0    ;ymm2: v3, v3, v3, v3
vunpckhpd ymm3,ymm7,ymm7
vperm2f128 ymm3,ymm3,ymm3,17    ;ymm3: v4, v4 v4, v4
```

7.3.3 Transpozícia matice - SSE

```
;xmm1: m14, m13, m12, m11
;xmm2: m24, m23, m22, m21
;xmm3: m34, m33, m32, m31
;xmm4: m44, m43, m42, m41
```

```
movups xmm0, xmm3       ;xmm0: m34, m33, m32, m31
punpckldq xmm3, xmm4    ;xmm3: m42, m32, m41, m31
punpckhdq xmm0, xmm4    ;xmm0: m44, m34, m43, m33
movups xmm4, xmm1       ;xmm4: m14, m13, m12, m11
punpckldq xmm1, xmm2    ;xmm1: m22, m12, m21, m11
punpckhdq xmm4, xmm2    ;xmm4: m24, m14, m23, m13
movups xmm2, xmm1       ;xmm2: m14, m13, m12, m11
punpcklqdq xmm1, xmm3   ;xmm1: m41, m31, m21, m11
punpckhqdq xmm2, xmm3   ;xmm2: m42, m32, m22, m12
movups xmm3, xmm4       ;xmm3: m24, m14, m23, m13
punpcklqdq xmm3, xmm0   ;xmm3: m43, m33, m23, m13
punpckhqdq xmm4, xmm0   ;xmm4: m44, m34, m24, m14
```

7.3.4 Transpozícia matice - AVX

```
;ymm0: m14, m13, m12, m11
;ymm1: m24, m23, m22, m21
;ymm2: m34, m33, m32, m31
;ymm3: m44, m43, m42, m41

vunpcklpd ymm4, ymm0, ymm1    ;ymm4: m22, m12, m21, m11
vunpckhpd ymm0, ymm0, ymm1    ;ymm0: m24, m14, m23, m13
vunpcklpd ymm1, ymm2, ymm3    ;ymm1: m42, m32, m41, m31
vunpckhpd ymm2, ymm2, ymm3    ;ymm2: m44, m34, m43, m33

vperm2f128 ymm3, ymm4, ymm1, 32   ;ymm3: m11, m21, m31, m41
vperm2f128 ymm1, ymm4, ymm1, 49   ;ymm1: m12, m22, m32, m42
vperm2f128 ymm4, ymm0, ymm2, 32   ;ymm4: m13, m23, m33, m43
vperm2f128 ymm2, ymm0, ymm2, 49   ;ymm2: m14, m24, m34, m44
```

7.3.5 Násobenie zľava - SSE

Nasledujúce ukážky optimalizovaných zdrojových kódov zobrazujú postup násobenia vektora 1x4 maticou 4x4 vo variantách

- transpozície vektora pomocou shufps inštrukcií,
- transpozície vektora pomocou unpcklps/unpckhps inštrukcií,
- transpozície matice s následným využitím haddps inštrukcií.

Pre všetky varianty bude nasledujúca časť kódu rovnaká a bude vyzerať nasledovne:

```
;esi: ukazateľ na vstupný vektor
;edx: ukazateľ na maticu
;edi: ukazateľ na výstupný vektor

mov eax, 16    ;eax: 16
movups xmm0, [esi]    ;xmm0: v4, v3, v2, v1
movups xmm4, [edx]    ;xmm4: m14, m13, m12, m11
add edx, eax    ;posun ukazateľa na maticu
movups xmm5, [edx]    ;xmm5: m24, m23, m22, m21
add edx, eax    ;posun ukazateľa na maticu
movups xmm6, [edx]    ;xmm6: m34, m33, m32, m31
add edx, eax    ;posun ukazateľa na maticu
movups xmm7, [edx]    ;xmm7: m44, m43, m42, m41
```

Výpočet - prvá varianta:

```
movups xmm1, xmm0    ;xmm1: v4, v3, v2, v1
shufps xmm1, xmm1, 0x00    ;xmm1: v1, v1, v1, v1
mulps xmm1, xmm4    ;xmm1: v1*m14, v1*m13, v1*m12, v1*m11
addps xmm2, xmm1    ;xmm2: xmm2 + xmm1
movups xmm1, xmm0    ;xmm1: v4, v3, v2, v1
shufps xmm1, xmm1, 0x55    ;xmm1: v2, v2, v2, v2
mulps xmm1, xmm5    ;xmm1: v2*m24, v1*m23, v1*m22, v1*m21
addps xmm2, xmm1    ;xmm2: xmm2 + xmm1
movups xmm1, xmm0    ;xmm1: v4, v3, v2, v1
shufps xmm1, xmm1, 0xAA    ;xmm1: v3, v3, v3, v3
mulps xmm1, xmm6    ;xmm1: v3*m34, v3*m33, v3*m32, v3*m31
addps xmm2, xmm1    ;xmm2: xmm2 + xmm1
movups xmm1, xmm0    ;xmm1: v4, v3, v2, v1
shufps xmm1, xmm1, 0xFF    ;xmm4: v4, v4, v4, v4
mulps xmm1, xmm7    ;xmm1: v4*m44, v4*m43, v4*m42, v4*m41
addps xmm2, xmm1    ;xmm2: xmm2 + xmm1
movups [edi], xmm2    ;uložíme výsledok, ktorý je v xmm2
```

Výpočet - druhá varianta:

```
movups xmm1, xmm0    ;xmm1: v4, v3, v2, v1
unpcklps xmm1, xmm1    ;xmm1: v2, v2, v1, v1
movups xmm2, xmm1    ;xmm2: v4, v3, v2, v1
unpcklps xmm1, xmm1    ;xmm1: v1, v1, v1, v1
```

```
unpckhps xmm2, xmm2     ;xmm2: v2, v2, v2, v2
mulps xmm1, xmm4     ;xmm1: v1*m11,v1*m12,v1*m13,v1*m14
addps xmm3, xmm1     ;xmm3: xmm3 + xmm1
mulps xmm2, xmm5     ;xmm2: v2*m21,v2*m22,v2*m23,v2*m24
addps xmm3, xmm2     ;xmm3: xmm3 + xmm2

movups xmm1, xmm0     ;xmm1: v4, v3, v2, v1
unpckhps xmm1, xmm1     ;xmm1: v4, v4, v3, v3
movups xmm2, xmm1     ;xmm2: v4, v4, v3, v3
unpcklps xmm1, xmm1     ;xmm1: v3, v3, v3, v3
unpckhps xmm2, xmm2     ;xmm2: v4, v4, v4, v4
mulps xmm1, xmm6     ;xmm1: v3*m31,v3*m32,v3*m33,v3*m34
addps xmm3, xmm1     ;xmm3: xmm3 + xmm1
mulps xmm2, xmm7     ;xmm2: v4*m41,v4*m42,v4*m43,v4*m44
addps xmm3, xmm2     ;xmm3: xmm3 + xmm2
movups [edi], xmm3     ;uložíme výsledok, ktorý je v xmm3
```

Výpočet - tretia varianta:

```
movups xmm1, xmm6
punpckldq xmm6, xmm7
punpckhdq xmm1, xmm7
movups xmm7, xmm4
punpckldq xmm4, xmm5
punpckhdq xmm7, xmm5
movups xmm5, xmm4
punpcklqdq xmm4, xmm6     ;xmm4: m41, m31, m21, m11
punpckhqdq xmm5, xmm6     ;xmm5: m42, m32, m22, m12
movups xmm6, xmm7
punpcklqdq xmm6, xmm1     ;xmm6: m43, m33, m23, m13
punpckhqdq xmm7, xmm1     ;xmm7: m44, m34, m24, m14

mulps xmm4, xmm0     ;xmm4: m41*v4, m31*v3, m21*v2, m11*v1
mulps xmm5, xmm0     ;xmm5: m42*v4, m32*v3, m22*v2, m12*v1
mulps xmm6, xmm0     ;xmm6: m43*v4, m33*v3, m23*v2, m13*v1
mulps xmm7, xmm0     ;xmm7: m44*v4, m34*v3, m24*v2, m14*v1
haddps xmm4, xmm5
haddps xmm6, xmm7
haddps xmm4, xmm6
```

```
addps xmm3, xmm4
movups [edi], xmm3    ;uložíme výsledok, ktorý je v xmm3
```

7.3.6 Násobenie zľava - AVX

Pri výpočte násobenia zľava využijeme dve varianty transpozície vektora (kap. 7.3.4).

Výpočet - prvá varianta:

```
;esi: ukazateľ na vstupný vektor
;edx: ukazateľ na maticu
;edi: ukazateľ na výstupný vektor

mov eax,32    ;eax: 32
vmovupd ymm0,[edx]    ;ymm0: m14, m13, m12, m11
add edx,eax    ;posun ukazateľa na maticu
vmovupd ymm1,[edx]    ;ymm1: m24, m23, m22, m21
add edx,eax
vmovupd ymm2,[edx]    ;ymm2: m34, m33, m32, m31
add edx,eax
vmovupd ymm3,[edx]    ;ymm3: m44, m43, m42, m41

vbroadcastsd ymm7, [esi]    ;ymm7: v1, v1, v1, v1
add esi,8    ;posun ukazateľa na vstupný vektor
vmulpd ymm0,ymm0,ymm7    ;násobenie riadku matice s vektorom
vaddpd ymm6,ymm6,ymm0    ;výsledok predošlého kroku pripočítaný k ymm6

vbroadcastsd ymm7, [esi]    ;ymm7: v2, v2, v2, v2
add esi,8
vmulpd ymm1,ymm1,ymm7
vaddpd ymm6,ymm6,ymm1

vbroadcastsd ymm7, [esi]    ;ymm7: v3, v3, v3, v3
add esi,8
vmulpd ymm2,ymm2,ymm7
vaddpd ymm6,ymm6,ymm2

vbroadcastsd ymm7, [esi]    ;ymm7: v4, v4 v4, v4
vmulpd ymm3,ymm3,ymm7
```

```
vaddpd ymm6,ymm6,ymm3

vmovupd [edi],ymm6    ;uložíme výsledok, ktorý je v ymm6
```

Výpočet - druhá varianta:

```
;esi: ukazateľ na vstupný vektor
;edx: ukazateľ na maticu
;edi: ukazateľ na výstupný vektor

vmovupd ymm0,[edx]    ;ymm0: m14, m13, m12, m11
add edx,eax
vmovupd ymm1,[edx]    ;ymm1: m24, m23, m22, m21
add edx,eax
vmovupd ymm2,[edx]    ;ymm2: m34, m33, m32, m31
add edx,eax
vmovupd ymm3,[edx]    ;ymm3: m44, m43, m42, m41
vmovupd ymm7,[esi]    ;ymm7: v4, v3, v2, v1

vunpcklpd ymm4,ymm7,ymm7    ;ymm4: v3, v3, v1, v1
vperm2f128 ymm4,ymm4,ymm4,0    ;ymm4: v1, v1, v1, v1
vmulpd ymm0, ymm0, ymm4    ;násobenie vektora maticou
vaddpd ymm6, ymm6, ymm0    ;pričítanie predošlého kroku k ymm6

vunpckhpd ymm4,ymm7,ymm7    ;ymm4: v4, v4, v2, v2
vperm2f128 ymm4,ymm4,ymm4,0    ;ymm4: v2, v2, v2, v2
vmulpd ymm1, ymm1, ymm4
vaddpd ymm6, ymm6, ymm1

vunpcklpd ymm4,ymm7,ymm7    ;ymm4: v3, v3, v1, v1
vperm2f128 ymm4,ymm4,ymm4,17    ;ymm4: v3, v3, v3 v3
vmulpd ymm2, ymm2, ymm4
vaddpd ymm6, ymm6, ymm2

vunpckhpd ymm4,ymm7,ymm7    ;ymm4: v4, v4, v2, v2
vperm2f128 ymm4,ymm4,ymm4,17    ;ymm4: v4, v4, v4, v4
vmulpd ymm3, ymm3, ymm4
vaddpd ymm6, ymm6, ymm3
```

```
vmovupd [edi],ymm6    ;uložíme výsledok, ktorý je v ymm6
```

7.3.7 Násobenie zľava - FMA

Pri využití inštrukcie **vfmadd231pd** (kap. 7.2) bude zdrojový kód výpočtu násobenia zľava pomocou FMA3 inštrukcií takmer zhodný s kódom z predošlej kapitoly. Naplnenie registrov vstupnými hodnotami a transpozícia vektora ostane bez zmeny. Jedinou zmenou bude zlúčenie inštrukcií násobenia (vmulpd) a sčítania (vaddpd) do jedného kroku inštrukciou vfmadd231pd.

Kroky

```
vmulpd ymm0, ymm0, ymm4
vaddpd ymm6, ymm6, ymm0
```

sa nahradia krokom

```
vfmadd231pd ymm6, ymm0, ymm4
```

čím sa kód skráti celkovo o 4 inštrukcie.

7.3.8 Násobenie sprava - SSE

Počiatočné naplnenie registrov hodnotami matice a vektora bude rovnaké, ako v kapitole násobenia zľava (kap. 7.3.5). Samotný výpočet násobenia matice vektorom bude vo variantách

- s využitím inštrukcií haddps,
- s využitím transpozície medzivýslednej matice po prvotnom násobení,
- s využitím inštrukcií shufps a unpcklps/unpckhps.

Výpočet - prvá varianta:

```
mulps xmm4,xmm0    ;xmm4: v4*m14, v3*m13, v2*m12, v1*m11
mulps xmm5,xmm0    ;xmm5: v4*m24, v3*m23, v2*m22, v1*m21
mulps xmm6,xmm0    ;xmm6: v4*m34, v3*m33, v2*m32, v1*m31
mulps xmm7,xmm0    ;xmm7: v4*m44, v3*m43, v2*m42, v1*m41

haddps xmm4,xmm5
haddps xmm6,xmm7
haddps xmm4,xmm6    ;xmm4: výsledný vektor
movups [edi],xmm4   ;uložíme výsledok, ktorý je v xmm4
```

Výpočet - druhá varianta:

```
mulps xmm4,xmm0    ;xmm4: v4*m14, v3*m13, v2*m12, v1*m11
mulps xmm5,xmm0    ;xmm5: v4*m24, v3*m23, v2*m22, v1*m21
mulps xmm6,xmm0    ;xmm6: v4*m34, v3*m33, v2*m32, v1*m31
mulps xmm7,xmm0    ;xmm7: v4*m44, v3*m43, v2*m42, v1*m41

;nasleduje transpozícia matice po prvotnom násobení
movups xmm1,xmm6
punpckldq xmm6,xmm7
punpckhdq xmm1,xmm7
movups xmm7,xmm4
punpckldq xmm4,xmm5
punpckhdq xmm7,xmm5
movups xmm5,xmm4
punpcklqdq xmm4,xmm6
punpckhqdq xmm5,xmm6
movups xmm6,xmm7
punpcklqdq xmm6,xmm1
punpckhqdq xmm7,xmm1

addps xmm4,xmm5
addps xmm4,xmm6
addps xmm4,xmm7
addps xmm2,xmm4
movups [edi],xmm2    ;uložíme výsledok, ktorý je v xmm2
```

Výpočet - tretia varianta:

```
mulps xmm4,xmm0
mulps xmm5,xmm0
mulps xmm6,xmm0
mulps xmm7,xmm0

movups xmm2,xmm4
unpcklps xmm2,xmm5
unpckhps xmm4,xmm5
movups xmm3,xmm6
unpcklps xmm3,xmm7
unpckhps xmm6,xmm7
movups xmm5,xmm2
```

```
shufps xmm5,xmm3,0x44
shufps xmm2,xmm3,0xEE
addps xmm2,xmm5
movups xmm7,xmm4
shufps xmm7,xmm6,0x44
shufps xmm4,xmm6,0xEE

addps xmm4,xmm7
addps xmm4,xmm2
addps xmm1,xmm4
movups [edi],xmm1    ;uložíme výsledok, ktorý je v xmm1
```

7.3.9 Násobenie sprava - AVX

Pri násobení matice vektorom s využitím výpočtovej jednotky AVX už nie je možné výsledný vektor vypočítať horizontálnym sčítaním, pretože Vex verzia inštrukcie haddps vykazuje odlišné správanie (viď. samostatnú prílohu k práci). Z tohto dôvodu musíme využiť alternatívne riešenie. Výpočet bude v jednej variante, kde využijeme transpozíciu medzimatice, ktorú dostaneme po prvotnom násobení vstupnej matice so vstupným vektorom. Následne po transpozícii využijeme vektorový súčet hodnôt a tým dostaneme výsledný vektor.

Výpočet:

```
;esi: ukazateľ na vstupný vektor
;edx: ukazateľ na maticu
;edi: ukazateľ na výstupný vektor

mov eax,32    ;eax: 32
vmovupd ymm0,[edx]    ;ymm0: m14, m13, m12, m11
add edx,eax    ;posun ukazateľa na maticu
vmovupd ymm1,[edx]    ;ymm1: m24, m23, m22, m21
add edx,eax
vmovupd ymm2,[edx]    ;ymm2: m34, m33, m32, m31
add edx,eax
vmovupd ymm3,[edx]    ;ymm3: m44, m43, m42, m41
vmovupd ymm7,[esi]    ;ymm7: v4, v3, v2, v1

vmulpd ymm0,ymm0,ymm7    ;ymm0: m14*v4, m13*v3, m12*v2, m11*v1
```

```
vmulpd ymm1,ymm1,ymm7    ;ymm1: m24*v4, m23*v3, m22*v2, m21*v1
vmulpd ymm2,ymm2,ymm7    ;ymm2 m34*v4, m33*v3, m32*v2, m31*v1
vmulpd ymm3,ymm3,ymm7    ;ymm3: m44*v4, m43*v3, m42*v2, m41*v1

;transpozícia medzimatice
vunpcklpd ymm4, ymm0, ymm1
vunpckhpd ymm0, ymm0, ymm1
vunpcklpd ymm1, ymm2, ymm3
vunpckhpd ymm2, ymm2, ymm3
vperm2f128 ymm3, ymm4, ymm1, 32
vperm2f128 ymm4, ymm4, ymm1, 49
vperm2f128 ymm1, ymm0, ymm2, 32
vperm2f128 ymm2, ymm0, ymm2, 49

;sčítanie transponovanej medzimatice
vaddpd ymm3,ymm3,ymm4
vaddpd ymm1,ymm1,ymm2
vaddpd ymm1,ymm1,ymm3
vaddpd ymm6,ymm6,ymm1
vmovupd [edi],ymm6    ;uložíme výsledok, ktorý je v ymm6
```

Keďže násobenie a sčítanie pri tomto type výpočtu nenasleduje bezprostredne po sebe, nie je možné využiť inštrukcie FMA3 pre túto variantu výpočtu. Ak by sme však aj použili upravený kód, logika výpočtu pomocou FMA3 by ostala zhodná ako pri násobení zľava, tj. nahradili by sme len inštrukcie násobenia a sčítania jednou inštrukciou. Kódy by preto boli v podstate rovnaké.

8 TESTOVANIE A VYHODNOTENIE VÝPOČTOV

Testy základných operácií s maticami a vektormi boli zamerané na porovnanie rýchlosti výpočtu výpočtových jednotiek SSE, AVX a FMA. Testy boli realizované pomocou príkladu diskrétnej kosínovej transformácie ako maticového počtu. Pri testovaní bol použitý rozmer matice 4096x4096, a teda vstupný a výstupný vektor mal rozmer buď 1x4096 alebo 4096x1, čo záviselo na type výpočtu (zľava/sprava). Každý test bol opakovaný 500krát a výsledná doba výpočtu je priemerom týchto hodnôt. Vysoký počet opakovaní je vhodné použiť z toho dôvodu, pretože výsledok jedného testu by nebol objektívny, čo je zapríčinené aktuálnym vyťažením procesoru.

V prvej časti testy používajú hodnoty v jednoduchej presnosti. Keďže je pri výpočtových jednotkách AVX a FMA možné použiť registre YMM o šírke 256 bitov, je možné využiť dvojnásobný počet hodnôt v jednom kroku ako pri SSE. Problémom však je, že pri 32bitovej verzii operačného systému je k dispozícii len 8 YMM registrov, čím nie je možné vykonať výpočty pre submatice, pretože tento počet registrov nie je dostačujúci pre vykonávanie potrebných operácií. Práve počítač, na ktorom boli zdrojové kódy testované, používal 32bitový operačný systém. Z toho dôvodu boli testy prvej časti zamerané na porovnanie deštruktívneho operandu pri SSE a nedeštruktívneho operandu pri AVX a FMA, a teda pracovalo sa s XMM registrami.

Druhá časť testov bola zameraná na výpočty s hodnotami v dvojitej presnosti a využitie výpočtových jednotiek novej generácie. V tejto časti bolo zámerom práce využitie YMM registrov, keďže v prvej časti to nebolo možné. Keďže mnoho základných inštrukcií pracujúcich s SSE má svoju novú verziu aj pri AVX a FMA, bolo spočiatku zámerom práce vytvoriť podobné kódy pre SSE a pre AVX a FMA. Tu však nastali problémy, pretože niektoré spomínané základné inštrukcie pracujú pri AVX a FMA odlišným spôsobom, ako pri SSE. Z toho dôvodu nebolo možné vytvoriť podobné kódy. Dokonca mnoho inštrukcií pracuje tak odlišne, že konečným zámerom druhej časti bolo vytvorenie zdrojových kódov, ktoré budú zamerané na výpočtové jednotky novej generácie, a ktoré budú využívať nové inštrukcie, ktorými výpočtová jednotka SSE nedisponuje. Inštrukčné súbory pre výpočtové jednotky AVX a FMA poskytujú viac inštrukcií na realizáciu násobenia zľava, a preto sú testované funkcie na násobenie zľava v dvojitej presnosti vo viacerých variantách.

Doby trvania jednotlivých výpočtov sú uvedené v tabuľke 8.1 pre jednoduchú presnosť a v tabuľke 8.2 pre dvojitú presnosť.

Pri výpočtoch v jednoduchej presnosti sú zdrojové kódy tvorené tak, že pre všetky výpočtové jednotky (SSE, AVX a FMA) používajú rovnaký typ výpočtu. Jeden typ

Tab. 8.1: Výsledky testov pre jednoduchú presnosť

Násobenie zľava					
1. spôsob (shuffle)		**2. spôsob (unpack)**		**3. spôsob (transpozícia matice + haddps)**	
SSE	15,4549 ms	SSE	15,367 ms	SSE	19,0277 ms
AVX	15,1644 ms	AVX	15,2365 ms	AVX	18,9323 ms
FMA	15,1817 ms	FMA	15,2293 ms		
Násobenie sprava					
1. spôsob (haddps)		**2. spôsob (transpozícia po násobení)**		**doplnkový spôsob pre SSE (shuffle+unpack)**	
SSE	16,1882 ms	SSE	16,9233 ms	SSE	16,7279 ms
AVX	15,9886 ms	AVX	16,639 ms		

Tab. 8.2: Výsledky testov pre dvojitú presnosť

Násobenie zľava	
AVX (permutácie a unpack)	29,8377 ms
AVX (broadcast)	30,3197 ms
FMA (broadcast)	29,5674 ms
Násobenie sprava	
AVX (transpozícia matice po násobení a permutácie)	34,3564 ms

výpočtu (transpozícia matice a následné využitie haddps inštrukcií) nebol pri výpočtovej jednotke FMA možný realizovať, pretože podmienkou pre využitie FMA je, aby bezprostredne po sebe nasledovali operácie násobenia a sčítania. To isté platí aj pre násobenie sprava, kde sa pri jednotkách SSE a AVX porovnávajú dva rovnaké typy výpočtu, ktoré však nie je možné použiť pri výpočtovej jednotke FMA, pretože po násobení nasleduje transpozícia matice, a až potom násobenie, čiže nie je splnená podmienka.

Vyhodnotenie testovania výpočtov v dvojitej presnosti

Ako už bolo spomenuté vyššie, pri testovaní kódov v dvojitej presnosti nebolo možné vytvoriť veľký počet spôsobov výpočtov pre všetky jednotky. Preto bolo zámerom práce ukázať aspoň nejaké doby výpočtov jednotiek novej generácie, resp. priblížiť

hodnoty, v akých sa výpočty v dvojitej presnosti pohybujú. Zámerom tiež bolo využitie inštrukcií, ktoré boli pridané v inštrukčných súboroch AVX, AVX2 a FMA3, teda takých, ktorými výpočtová jednotka SSE nedisponuje. Najrýchlejší spôsob výpočtu násobenia zľava predviedla výpočtová jednotka FMA. Pri výpočte násobenia sprava sa potvrdilo, že transpozícia matice je výpočtovo náročná operácia.

Vyhodnotenie testovania výpočtov v jednoduchej presnosti

Násobenie zľava

1. spôsob (shuffle): doba trvania výpočtu je pri výpočtovej jednotke SSE pomalšia ako pri jednotkách AVX a FMA. To je spôsobené práve využitím nedeštruktívneho operandu, vďaka ktorému sa zdrojový kód skrátil. Výpočet pomocou AVX je nepatrne rýchlejší ako pri FMA, čo vedie k záveru, že teoretické hodnoty zrýchlenia doby výpočtu sa pri výpočtovej jednotke FMA nepotvrdili.

2. spôsob (unpack): výpočtová jednotka SSE je opäť najpomalšia pri tomto spôsobe výpočtu, avšak už nie príliš výrazne. Najrýchlejší výpočet predviedla jednotka FMA, ale výpočtový čas nie je výrazne.

3. spôsob (transp., horizontal add): mierne zrýchlenie výpočtu skrátením zdrojového kódu vďaka nedeštruktívnemu operandu poskytla výpočtová jednotka AVX. Toto zrýchlenie však nie je výrazné. Platí, že transpozícia matice je výpočtovo náročná operácia, a preto sú doby výpočtov vyššie ako pri prechádzajúcich spôsoboch výpočtov.

Násobenie sprava

1. spôsob SSE vs. AVX (horizontal add): výpočtová jednotka AVX pri využití tohto typu výpočtu urýchlila výpočet o niečo výraznejšie, ako tomu bolo pri násobení zľava.

2. spôsob SSE vs. AVX (transpozícia po násobení): hodnoty transpozície matice sú v tomto type násobenia výrazne lepšie, ako pri násobení zľava. Je to spôsobené aj tým, že inštrukcie haddps nie sú tak rýchle, ako by sa mohlo na prvý pohľad zdať. Samotný výpočet zvládla lepšie výpočtová jednotka AVX, čo bolo spôsobené opäť skrátením kódu s využitím nedeštruktívneho operandu.

Doplnkový spôsob pre SSE (shuffle vs. unpack): tento spôsob výpočtu bol výrazne pomalší, ako 1. spôsob, ale v porovnaní s 2. spôsobom výpočtu poskytuje

výpočtová jednotka SSE pomerne značné urýchlenie výpočtu.

Celkové vyhodnotenie testovania výpočtov v jednoduchej presnosti

Zdrojové kódy pre jednotky novej generácie boli podstatne kratšie, ako zdrojové kódy pre výpočtovú jednotku SSE, tj. na výpočet daných operácií bol potrebný nižší počet inštrukcií. Skrátanie kódu pri výpočtovej jednotke AVX a FMA poskytol hlavne nedeštruktívny operand a navyše pri jednotke FMA sa kód skrátil ešte viac oproti AVX využitím inštrukcií násobenia a sčítania v jednom kroku. Teoreticky by sa skrátením kódu malo dosiahnuť výrazne zrýchelnie výpočtovej doby pri jednotkách AVX a FMA oproti výpočtovej jednotke SSE, čo sa však nepotvrdilo. Skrátenie kódu nie je úmerné dobe výpočtu. Rozdiely hodnôt medzi výpočtovými jednotkami nie sú výrazné, čo odporuje tvrdeniu o teoreticky dvoj- a viacnásobnom zrýchlení. To je spôsobené tým, že každá inštrukcia má svoju odovzu a priepustnosť. Tieto veličiny však pre jednu inštrukciu môžu byť rôzne pri využití procesoru inej mikroarchitektúry. Inými slovami, konkrétna inštrukcia má inú odozvu a priepustnosť napr. v procesore mikroarchitektúry Sandy Bridge a inú v procesore mikroarchitektúry Haswell.

Hodnoty priepustnosti jednotlivých inštrukcií, ktoré udáva výrobca, sú pre použité výpočtové jednotky zhodné pri mikroarchitektúre Haswell, ktorá bola pri testoch využitá. Napr. inštrukcia haddps ma pre všetky výpočtové jednotky priepustnosť 2 operácie v jednom strojovom cykle procesoru, zatiaľ čo inštrukcie shufps a unpcklps/unpckhps majú priepustnosť 1. To znamená, že inštrukciou haddps by sme mali dosiahnuť rýchlejší výpočet, to však neplatí. Značné uzýchlenie neposkytuje ani výpočtová jednotka FMA. Inštrukcia vfmadd231ps má priepustnosť len 0.5, takže dokáže vykonať len polovicu inštrukcie v jednom strojovom cykle procesoru. Z toho vyplýva, že priepustnosť inštrukcií môže mať značný vplyv na výslednú dobu.

Parametre osobného počítača, na ktorom boli testované výpočty:

Operačný systém: Microsoft Windows 7 Home Edition, 32bitová verzia.
Procesor: Intel Core i5-4200U, mikroarchitektúra Haswell.
Frekvencia a počet jadier: 2 jadrá, každé 1,6 GHz.
Operačná pamäť: 4 GB.

9 ZÁVER

Cieľom práce bolo uskutočniť analýzu a následné porovnanie starších a nových vektorových jednotiek, ktoré sa používajú v súčasných moderných procesoroch, najmä vektorových jednotiek SSE, AVX a FMA, porovnanie spôsobu výpočtov týchto jednotiek a uvedenie ich prínosu.

Teoretická časť práce sa delí do piatich hlavných častí. Prvá časť predstavuje výpočtové systémy so zameraním na systém SIMD a následne poskytuje stručný historický prehľad vývoja vektorových výpočtových jednotiek. Druhá časť práce popisuje výpočtové jednotky a poskytuje priblíženie toho, akým spôsobom tieto výpočtové jednotky pracujú s dátami. Implementácia vektorových výpočtových jednotiek v mikroprocesoroch u firiem Intel a AMD je uvedná v tretej časti. Vo štvrtej časti sa práca zaoberá súbormi inštrukcií, ktoré využívajú vektorové výpočtové jednotky. Práca poskytuje zoznam jednotlivých inštrukčných súborov a ich krátky prehľad. Samotný zoznam inšturkcií pre každú inštrukčnú sadu poskytuje samostatná príloha k práci, ktorá tiež obsahuje popis správania sa týchto inštrukcií. Posledná časť predstavuje inštrukciu CPUID a jej možnosti, vďaka ktorým je pomocou tejto inštrukcie možné zisťovať informácie o procesore.

Praktická časť práce je venovaná porovnaniu výpočtových jednotiek staršej a novšej generácie. Práca poskytuje porovnanie výpočtov jednotlivých výpočtových jednotiek. Táto časť práce rozoberá realizáciu základných vektorových operácií s maticami a vektormi, najmä objasnenie spôsobu, akým pracuje pri realizácii týchto výpočtov procesor a čo je nutné si pri práci s inštrukciami uvedomiť.

Prínosom práce je porovnanie doby trvania operácií s maticami a vektormi pomocou jednotlivých výpočtových jednotiek. Na konkrétnych príkladoch maticového a vektorového výpočtu bolo poskytnuté porovnanie rýchlosti týchto jednotiek a tiež porovnanie spôsobov, akými výpočtové jednotky spracovávajú dáta. Očakávané teoretické hodnoty sa nezhodovali s hodnotami dosiahnutými po uskutočnení praktických testov, a teda boli vyvrátené tvrdenia výrobcu o niekoľkonásobnom zrýchlení výpočtových jednotiek novej generácie oproti výpočtovým jednotkám staršej generácie. Zistené hodnoty poukazujú na len mierny nárast rýchlosti nových výpočtových jednotiek.

Súčasťou práce sú dva teoretické úvody k laboratórnym úlohám uvedené v prílohách.

LITERATÚRA

[1] SIEWERT, S. *Using Intel® Streaming SIMD Extensions and Intel® Integrated Performance Primitives to Accelerate Algorithms* [online]. 2009, posledná aktualizácia 3. 11. 2014, [cit. 7. 11. 2014]. Dostupné z URL: <https://software.intel.com/en-us/articles/using-intel-streaming-simd-extensions-and-intel-integrated-performance-primi

[2] INTEL®. *Intel® 64 and IA-32 Architectures Software Developer's Manual: Volume 1: Basic Architecture* [online]. 1997, posledná aktualizácia 2014 [cit. 8. 11. 2014]. Dostupné z URL: <http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-1-manual.pdf>.

[3] LOMONT, Ch. *Introduction to Intel® Advanced Vector Extensions* [online]. 2011, posledná aktualizácia 7. 11. 2014, [cit. 9. 11. 2014]. Dostupné z URL:<https://software.intel.com/en-us/articles/introduction-to-intel-advanced-vector-extensions>.

[4] SCARPINO, M. *Crunching Numbers with AVX and AVX2* [online]. 20. 2. 2015, [cit. 25. 3. 2015]. Dostupné z URL:<http://www.codeproject.com/Articles/874396/Crunching-Numbers-with-AVX-and-AVX>.

[5] BALÍK, M. *Počítače a jejich periférie*. Prvé vydanie, Brno: Vysoké učení technické v Brně, Ústav telekomunikací, 2012. ISBN 978-80-214-4635-9.

[6] FOG, A. *The microarchitecture of Intel, AMD and VIA CPUs* [online]. 1996, posledná aktualizácia 7. 8. 2014, [cit. 24. 11. 2014]. Dostupné z URL:<http://www.agner.org/optimize/microarchitecture.pdf>.

[7] KANTER, D. *Intel's Sandy Bridge Microarchitecture* [online]. 25. 9. 2010, [cit. 4. 12. 2014]. Dostupné z URL:<http://www.realworldtech.com/sandy-bridge/6/>.

[8] KANTER, D. *Intel's Haswell CPU Microarchitecture* [online]. 13. 11. 2012, [cit. 4. 12. 2014]. Dostupné z URL:<http://www.realworldtech.com/haswell-cpu/4/>.

[9] Scali's OpenBlog™. *AMD Steamroller* [online]. 26. 9. 2012, [cit. 5. 12. 2014]. Dostupné z URL:<http://scalibq.wordpress.com/2012/09/26/amd-steamroller/>.

[10] WALTON, J. *The AMD Trinity Review (A10-4600M): A New Hope* [online]. 2012, [cit. 24. 11. 2014]. Dostupné z URL: <http://www.anandtech.com/show/5831/amd-trinity-review-a10-4600m-a-new-hope>.

[11] Intel®. *Processor Identification and the CPUID instruction* [online]. 2011, [cit. 20. 4. 2015]. Dostupné z URL:<http://web.archive.org/web/20110307080258/http://www.intel.com/Assets/PDF/appnote/241618.pdf>.

ZOZNAM SYMBOLOV, VELIČÍN A SKRATIEK

ADX – Multi-Precision Add-Carry Instruction Extensions

AGU – Address Generation Unit

ALU – Arithmetic Logic Unit

AMD – Advanced Micro Devices

APIC – Advanced Programmable Interrupt Controller

AVX – Advanced Vector Extension

BMI – Bit Manipulation Instruction

cALU – complex Arithmetic Logic Unit

CMT – Clustered Multi-Thread

CPL – Control Panel Aplet

CPU – Central Processing Unit

CPUID – Central Processing Unit Identification

CR4 – Control Register 4

EAX – Accumulator Register

EBP – Stack Base Pointer Register

EBX – Base Register

ECX – Counter Register

EDI – Destination Index Register

EDX – Data Register

ESI – Source Index Register

ESP – Stack Pointer Register

FMA – Fused Multiply-Add

FMAC – Fused Multiply-Accumulate

FP – Floating-Point

FPU – Floating-Point Unit

IA – Intel Architecture

JEU – Jump Execution Unit

L1 – Level 1

MCA – Machine Check Architecture

MCE – Machine Check Exception

MIMD – Multiple Instruction-Multiple Data

MISD – Multiple Instruction-Single Data

MMX – Multimedia Extension

MSR – Model-Specific Register

MXCSR – Multimedia Extensions Control and Status Register

PAT – Page Attribute Table

PBE – Pending Break Enable

PGE – Page Global Enable

SIMD – Single Instruction-Multiple Data

SISD – Single Instruction-Single Data

SMAP – Supervisor Mode Access Prevention

SMEP – Supervisor Mode Execution Prevention

SMX – Safer Mode Extenisions

SSE – Streaming SIMD Extension

SSSE – Suplemental Streaming SIMD Extension

TSC – Time Stamp Counter

TSX – Transactional Synchronization Extensions

uAGU – universal Address Gerenation Unit

vecALU – vector Arithmetic Logic Unit

VEX – Vector Extension

VLE – Vector Lenght Extension

VMX – Virtual Machine Extension

ZOZNAM PRÍLOH

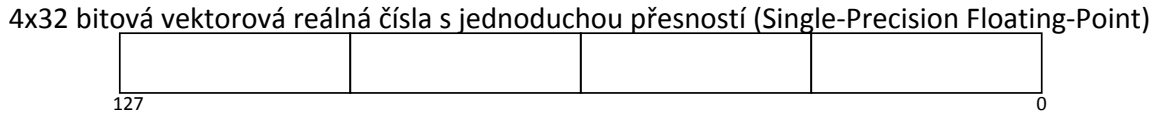
A POROVNÁNÍ VÝPOČTŮ VEKTOROVÝCH OPERACÍ POMOCÍ SSE

A.1 Teoretický úvod

A.1.1 Programové prostředí SSE

SSE (Streaming SIMD Extensions) je instrukční sada SIMD (Single Instruction - Multiple Data). Už z jejího názvu je patrné, že jedna SSE instrukce dokáže pracovat s více daty současně.

Výpočetní jednotka SSE pracuje se 128bitovými registry XMM0 až XMM7 v chráněném režimu procesoru, resp. XMM0 až XMM15 v tzv. dlouhém režimu procesoru. Registry pracují s datovými typy. Instrukční sada SSE2 přidává nové datové typy k předešlým, které pracují s velikostí 128 bitů. Předcházející instrukční sada SSE pracovala pouze s jedním datovým typem, který měl velikost 128 bitů a pracoval vektorově se čtyřmi reálnými čísly s jednoduchou přesností (obr. A.1).

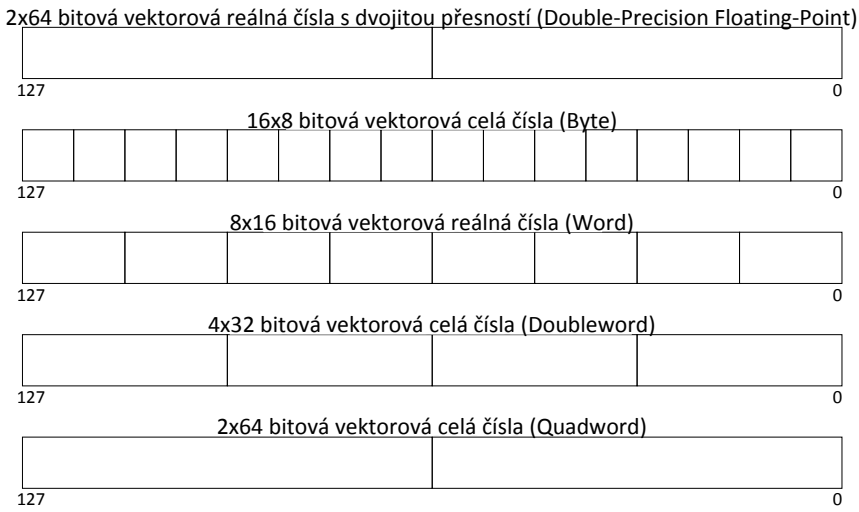


Obr. A.1: 128bitový datový typ technologie SSE

Datové typy, které přináší SSE2 jsou (obr. A.2)

- 2x64bitová vektorová reálná čísla s dvojitou přesností (Double-Precision Floating-Point),
- 16x8bitová vektorová celá čísla (Byte),
- 8x16bitová vektorová celá čísla (Word),
- 4x32bitová vektorová celá čísla (Doubleword),
- 2x64bitová vektorová celá čísla (Quadword).

Výpočetní jednotka SSE při výpočtech používá tzv. destruktivní operand, tj. po vykonání operace s hodnotama XMM registrů (např. součet) je výsledek uložen do jednoho z operandů, a tak je jeho předchozí hodnota přepsána novou hodnotou. Proto je nezbytné při práci s instrukcemi SSE brát na tohle ohled.



Obr. A.2: Datové typy SSE2

A.2 Vzorový příklad, násobení vektoru s maticí (násobení zleva)

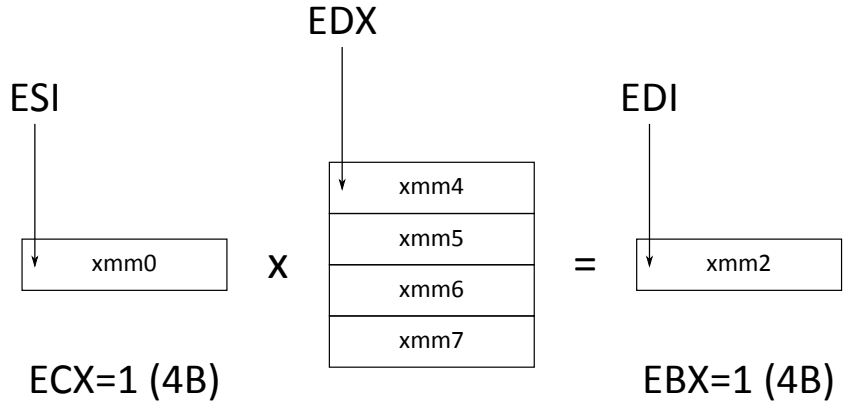
V tomto příkladu bude provedeno maticové násobení mezi vstupním vektorem a maticí s využitím instrukcí SSE a datových typů, pracujících s reálnyma číslama v dvojité přesnosti.

Kromě XMM registrů využijeme také standardní registry procesoru EDX, ESI, EDI, ECX a EBX. Registry ECX a EBX obsahují délku vstupního a výstupního vektoru. Registry ESI, EDX a EDI jsou použity jako ukazatelé. Registr ESI ukazuje na aktuální prvek vstupního vektoru, tj. prvek který se bude zpracovávat. Registr EDX ukazuje na aktuální prvek matice a registr EDI na aktuální prvek ve výstupním vektoru.

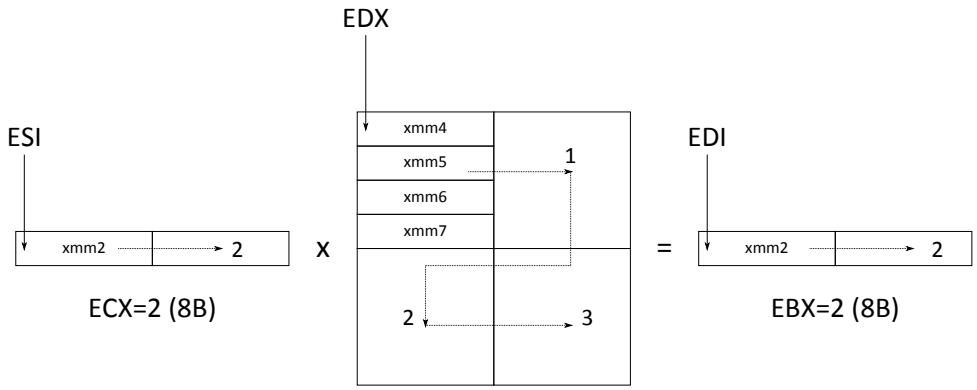
Při návrhu algoritmu, který bude realizovat násobení, je třeba jsi uvědomit, že SSE registry jsou vektorové (datové typy packed). V našem případě to znamená, že jednotlivé prvky jsou zpracovávány po čtyřech při použití hodnot v jednoduché přesnosti (32 bitů).

Popis použitých registrů je graficky znázorněn na obrázku A.3. Za upozornění stojí, že délky obou vektorů jsou udávány v násobcích čísla 4.

Na obrázku A.4 je zobrazena situace při násobení vektoru s maticí o větších rozměrech. Obrázek naznačuje posuvy pro matici o rozměrech 8x8, kterou musíme rozdělit na submatice.



Obr. A.3: Maticové násobení, využití registrů SSE



Obr. A.4: Maticové násobení, využití registrů SSE pro větší rozměry matic

A.2.1 Zdrojové kódy

V tomto příkladu je násoben vstupní vektor s maticí jakýchkoli rozměrů. Rozměry jsou ale v násobcích 4 bytů. Pokud má tedy matice rozměr 4x4 bytů, tak registry ECX a EBX budou mít hodnoty rovny 1. Výpočet je realizován třemi způsoby:

- Dva kódy pro transpozici vstupního vektoru využitím jiných instrukcí samotné transpozice.
- Kód s transpozicí matice na násobení zprava.

Výpočet - první varianta:

```
vxm_sse1:
push dword ebp
mov dword ebp,esp
```

```
sub esp,4       ;rezerva pro jednu lokální proměnnou 4B
mov dword [ebp - 4],3       ;nastavení hodnoty lokální proměnné

mov eax,16
mul dword [ebp + 24]

push eax
mul dword [ebp - 4]
mov dword [ebp - 4],eax
pop eax

mov esi,[ebp + 12]          ;ukazatel na pole x
mov ecx,[ebp + 20]       ;délka vstupního vektoru (v nas. 4B)
mov edx,[ebp + 8]           ;ukazatel na pole M

.invec1:    ;začátek prvního cyklu
movups xmm0,[esi]
mov edi,[ebp + 16]          ;ukazatel na pole y
mov ebx,[ebp + 24]       ;délka výstupního vektoru (pole y)

.radmat1:    ;začátek druhého cyklu
movups xmm4,[edx]
add edx,eax
movups xmm5,[edx]
add edx,eax
movups xmm6,[edx]
add edx,eax
movups xmm7,[edx]
movups xmm2,[edi]

movups xmm1,xmm0
shufps xmm1,xmm1,0x00
mulps xmm1,xmm4
addps xmm2,xmm1
movups xmm1,xmm0
shufps xmm1,xmm1,0x55
mulps xmm1,xmm5
```

```
addps xmm2,xmm1
movups xmm1,xmm0
shufps xmm1,xmm1,0xAA
mulps xmm1,xmm6
addps xmm2,xmm1
movups xmm1,xmm0
shufps xmm1,xmm1,0xFF
mulps xmm1,xmm7
addps xmm2,xmm1
movups [edi],xmm2

add edi,16
sub edx,dword [ebp - 4]
add edx,16
dec ebx
jnz .radmat1    ;konec prvního cyklu

add esi,16
add edx,dword [ebp - 4]
dec ecx
jnz .invec1    ;konec druhého cyklu

mov dword esp,ebp
pop dword ebp
ret 20
```

Výpočet - druhá varianta:

```
vxm_sse2:
push dword ebp
mov dword ebp,esp

sub esp,4    ;rezerva pro jednu lokální proměnnou 4B
mov dword [ebp - 4],3    ;nastavení hodnoty lokální proměnné

mov eax,16
mul dword [ebp + 24]
```

```
push eax
mul dword [ebp - 4]
mov dword [ebp - 4],eax
pop eax

mov esi,[ebp + 12]      ;ukazatel na pole x
mov ecx,[ebp + 20]    ;délka vstupního vektoru (v nas. 4B)
mov edx,[ebp + 8]       ;ukazatel na pole M

.invec2:
movups xmm0,[esi]
mov edi,[ebp + 16]      ;ukazatel na pole y
mov ebx,[ebp + 24]    ;délka výstupního vektoru (pole y)
.radmat2:
movups xmm4,[edx]
add edx,eax
movups xmm5,[edx]
add edx,eax
movups xmm6,[edx]
add edx,eax
movups xmm7,[edx]
movups xmm3,[edi]

movups xmm1,xmm0
unpcklps xmm1, xmm1
movups xmm2,xmm1
unpcklps xmm1, xmm1
unpckhps xmm2, xmm2
mulps xmm1, xmm4
addps xmm3, xmm1
mulps xmm2, xmm5
addps xmm3, xmm2

movups xmm1,xmm0
unpckhps xmm1, xmm0
movups xmm2,xmm1
unpcklps xmm1, xmm1
unpckhps xmm2, xmm2
```

```
mulps xmm1, xmm6
addps xmm3, xmm1
mulps xmm2, xmm7
addps xmm3, xmm2
movups [edi],xmm3

add edi,16
sub edx,dword [ebp - 4]
add edx,16
dec ebx
jnz .radmat2

add esi,16
add edx,dword [ebp - 4]
dec ecx
jnz .invec2

mov dword esp,ebp
pop dword ebp
ret 20
```

Výpočet - třetí varianta:

```
vxm_sse3:
push dword ebp
mov dword ebp,esp

sub esp,4    ;rezerva pro jednu lokální proměnnou 4B
mov dword [ebp - 4],3    ;nastavení hodnoty lokální proměnné

mov eax,16
mul dword [ebp + 24]

push eax
mul dword [ebp - 4]
mov dword [ebp - 4],eax
pop eax
```

```
mov esi,[ebp + 12]       ;ukazatel na pole x
mov ecx,[ebp + 20]    ;délka vstupního vektoru (v nas. 4B)
mov edx,[ebp + 8]        ;ukazatel na pole M

.invec3:
movups xmm0,[esi]
mov edi,[ebp + 16]       ;ukazatel na pole y
mov ebx,[ebp + 24]    ;délka výstupního vektoru (pole y)
.radmat3:
movups xmm4,[edx]
add edx,eax
movups xmm5,[edx]
add edx,eax
movups xmm6,[edx]
add edx,eax
movups xmm7,[edx]
movups xmm3,[edi]

movups xmm1,xmm6
punpckldq xmm6,xmm7
punpckhdq xmm1,xmm7
movups xmm7,xmm4
punpckldq xmm4,xmm5
punpckhdq xmm7,xmm5
movups xmm5,xmm4
punpcklqdq xmm4,xmm6
punpckhqdq xmm5,xmm6
movups xmm6,xmm7
punpcklqdq xmm6,xmm1
punpckhqdq xmm7,xmm1

mulps xmm4,xmm0
mulps xmm5,xmm0
mulps xmm6,xmm0
mulps xmm7,xmm0
haddps xmm4,xmm5
haddps xmm6,xmm7
haddps xmm4,xmm6
```

```
addps xmm3,xmm4
movups [edi],xmm3

add edi,16
sub edx,dword [ebp - 4]
add edx,16
dec ebx
jnz .radmat3

add esi,16
add edx,dword [ebp - 4]
dec ecx
jnz .invec3

mov dword esp,ebp
pop dword ebp
ret 20
```

Po linkování a kompilování kódu do dynamické knihovny využijeme program Matlab pro otestování správnosti výpočtu a pro zjištění doby jednotlivých výpočtů. V Matlabu vytvoříme skript, který bude volat funkce z dynamické knihovny. Jako vzorový příklad využijeme diskrétní kosinovou transformaci (DCT).

Skript v Matlabu:

```
% Uzavreni vsech grafu a smazani vsech promennych
clear all; clc

f1=1000;
f2=1100;
fs=8000;

N=4096; % řád matice a vektoru
n=0:N-1;
k=0:N-1;

% definice signalu; definice v jednoduche presnosti
x=single(0.7*cos(2*pi*f1/fs*(0:N-1))+0.3*cos(2*pi*f2/fs*(0:N-1)));

%výpočet DCT verze IV; definice v jednoduche presnosti
```

```
M=single(cos(pi/N*(n'+1/2)*(k+1/2)));

disp('Výpočet v Matlabu:')
tic
x*M;
toc
y=x*M;

% počet opakování volání funkce
Np=100;

%v ykresleni koeficientu DCT po normalizaci
plot(y * sqrt(2/N),'r') %normalizace DCT
hold on

% nacteni knihovny
hfile1 = 'test.h';
[notfound,warnings]=loadlibrary('test.dll', hfile1,'mfilename','test_mx');
pM = libpointer('singlePtr', M);
px = libpointer('singlePtr', x);

% volani funkce vxm_sse1
disp('Výpočet DCT v ASM pomocí SSE shuffle:')
td=0;
for i=1:Np
y0=zeros(1,N); py0 = libpointer('singlePtr', y0);
tic;
calllib('test', 'vxm_sse1', pM, px, py0, length(x)/4, length(y)/4);
td=td+toc/Np;
end
disp(num2str(td,6))
y_sse1=get(py0, 'Value');
plot(y_sse1 * sqrt(2/N),'g')

% volani funkce vxm_sse2
disp('Výpočet DCT v ASM pomocí SSE unpack:')
td=0;
for i=1:Np
```

```
y0=zeros(1,N); py0 = libpointer('singlePtr', y0);
tic;
calllib('test', 'vxm_sse2', pM, px, py0, length(x)/4, length(y)/4);
td=td+toc/Np;
end
disp(num2str(td,6))
y_sse2=get(py0, 'Value');
plot(y_sse2 * sqrt(2/N),'m')

% volani funkce vxm_sse3
disp('Výpočet DCT v ASM pomocí SSE transpozice a haddps:')
td=0;
for i=1:Np
y0=zeros(1,N); py0 = libpointer('singlePtr', y0);
tic;
calllib('test', 'vxm_sse3', pM, px, py0, length(x)/4, length(y)/4);
td=td+toc/Np;
end
disp(num2str(td,6))
y_sse3=get(py0, 'Value');
plot(y_sse3 * sqrt(2/N),'k')

hold off

unloadlibrary test
legend('DCT IV jako maticové násobení v Matlabu','DCT IV pomocí funkce
vxm sse1','DCT IV pomocí funkce vxm sse2','DCT IV pomocí funkce vxm
sse3',1)
```

A.3 Vzorový příklad, násobení matice vektorem (násobení zprava)

Změnou při tomto typu výpočtu budou použité instrukce a změna posunu ukazatelů vstupního a výstupního vektoru. Samotný výpočet je realizován opět třemi způsoby:

- Kód s využitím instrukcí tzv. horizontálního sčítání.
- Dva kódy pro transpozici matice po násobení.

A.3.1 Zdrojové kódy

Výpočet - první varianta:

```
mxv_sse4:
push dword ebp
mov dword ebp,esp

sub esp,4
mov dword [ebp - 4],3

mov eax,16
mul dword [ebp + 24]

push eax
mul dword [ebp - 4]
mov dword [ebp - 4],eax
pop eax

mov edi,[ebp + 16]       ;ukazatel na pole y (vystupny vektor)
mov ebx,[ebp + 24]    ;délka vstupního vektoru (v nas. 4B)
mov edx,[ebp + 8]       ;ukazatel na pole M

.invec4:
mov esi,[ebp + 12]       ;ukazatel na pole x
mov ecx,[ebp + 20]    ;délka výstupního vektoru (pole y)

.radmat4:
movups xmm4,[edx]
add edx,eax
movups xmm5,[edx]
add edx,eax
movups xmm6,[edx]
add edx,eax
movups xmm7,[edx]
movups xmm2,[edi]
movups xmm0,[esi]

mulps xmm4,xmm0
mulps xmm5,xmm0
```

```
mulps xmm6,xmm0
mulps xmm7,xmm0
haddps xmm4,xmm5
haddps xmm6,xmm7
haddps xmm4,xmm6
addps xmm2,xmm4
movups [edi],xmm2

add esi,16
sub edx,dword [ebp - 4]
add edx,16
dec ecx
jnz .radmat4

add edi,16
add edx,dword [ebp - 4]
dec ebx
jnz .invec4

mov dword esp,ebp
pop dword ebp
ret 20
```

Výpočet - druhá varianta:

```
mxv_sse5:
push dword ebp
mov dword ebp,esp

sub esp,4    ;rezerva pro jednu lokální proměnnou 4B
mov dword [ebp - 4],3    ;nastavení hodnoty lokální proměnné

mov eax,16
mul dword [ebp + 24]

push eax
mul dword [ebp - 4]
mov dword [ebp - 4],eax
```

```
pop eax
mov edi,[ebp + 16]      ;ukazatel na pole y
mov ebx,[ebp + 24]    ;délka vstupního vektoru (v nas. 4B)
mov edx,[ebp + 8]       ;ukazatel na pole M

.invec5:
;movups xmm0,[esi]
mov esi,[ebp + 12]      ;ukazatel na pole x
mov ecx,[ebp + 20]    ;délka výstupního vektoru (pole y)

.radmat5:
movups xmm4,[edx]
add edx,eax
movups xmm5,[edx]
add edx,eax
movups xmm6,[edx]
add edx,eax
movups xmm7,[edx]
movups xmm2,[edi]
movups xmm0,[esi]

mulps xmm4,xmm0
mulps xmm5,xmm0
mulps xmm6,xmm0
mulps xmm7,xmm0

movups xmm1,xmm6
punpckldq xmm6,xmm7
punpckhdq xmm1,xmm7
movups xmm7,xmm4
punpckldq xmm4,xmm5
punpckhdq xmm7,xmm5
movups xmm5,xmm4
punpcklqdq xmm4,xmm6
punpckhqdq xmm5,xmm6
movups xmm6,xmm7
punpcklqdq xmm6,xmm1
punpckhqdq xmm7,xmm1
```

```
addps xmm4,xmm5
addps xmm4,xmm6
addps xmm4,xmm7
addps xmm2,xmm4
movups [edi],xmm2

add esi,16
sub edx,dword [ebp - 4]
add edx,16
dec ecx
jnz .radmat5

add edi,16
add edx,dword [ebp - 4]
dec ebx
jnz .invec5

mov dword esp,ebp
pop dword ebp
ret 20
```

Výpočet - třetí varianta:

```
mxv_sse6:
push dword ebp
mov dword ebp,esp

sub esp,4    ;rezerva pro jednu lokální proměnnou 4B
mov dword [ebp - 4],3    ;nastavení hodnoty lokální proměnné

mov eax,16
mul dword [ebp + 24]

push eax
mul dword [ebp - 4]
mov dword [ebp - 4],eax
pop eax
```

```
mov edi,[ebp + 16]      ;ukazatel na pole y
mov ebx,[ebp + 24]    ;délka vstupního vektoru (v nas. 4B)
mov edx,[ebp + 8]       ;ukazatel na pole M

.invec6:
;movups xmm0,[esi]
mov esi,[ebp + 12]      ;ukazatel na pole x
mov ecx,[ebp + 20]    ;délka výstupního vektoru (pole y)

.radmat6:
movups xmm4,[edx]
add edx,eax
movups xmm5,[edx]
add edx,eax
movups xmm6,[edx]
add edx,eax
movups xmm7,[edx]
movups xmm1,[edi]
movups xmm0,[esi]

mulps xmm4, xmm0
mulps xmm5, xmm0
mulps xmm6, xmm0
mulps xmm7, xmm0

movups xmm2,xmm4
unpcklps xmm2,xmm5
unpckhps xmm4,xmm5
movups xmm3,xmm6
unpcklps xmm3,xmm7
unpckhps xmm6,xmm7
movups xmm5,xmm2
shufps xmm5,xmm3,0x44
shufps xmm2,xmm3,0xEE
addps xmm2,xmm5
movups xmm7,xmm4
shufps xmm7,xmm6,0x44
```

```
shufps xmm4,xmm6,0xEE
addps xmm4,xmm7

addps xmm4,xmm2
addps xmm1,xmm4
movups [edi],xmm1

add esi,16
sub edx,dword [ebp - 4]
add edx,16
dec ecx
jnz .radmat6

add edi,16
add edx,dword [ebp - 4]
dec ebx
jnz .invec6

mov dword esp,ebp
pop dword ebp
ret 20
```

V Matlabu se skript bude téměř shodovat. Jedinou změnou budou názvy volaných funkcí a po definici matice a vektoru je nutné je oba transponovat příkazem x=x' a M=M'.

B POROVNÁNÍ VÝPOČTŮ VEKTOROVÝCH OPERACÍ POMOCÍ AVX A FMA3

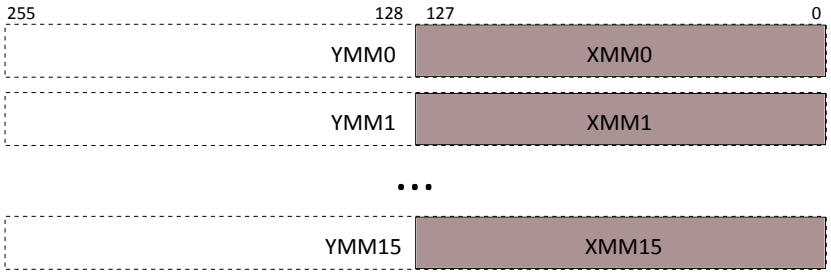
B.1 Teoretický úvod

B.1.1 Programové prostředí AVX a FMA

AVX (Advanced Vector Extensions) a FMA3 jsou instrukční sady SIMD, které přináší několik zásadních změn naproti SSE:

- AVX zavádí nové 256bitové registry YMM0 až YMM7 (YMM0 až YMM15 v 64bitové verzi OS), na kterých pracují dva nové datové typy.
- AVX používa nové kódovací schéma VEX-prefix (Vector Extension-prefix), díky čemu je možné pracovat s tzv. nedestruktivním operandem.
- FMA3 poskytuje soubor instrukcí, které slučují operace násobení a sčítání, nebo násobení a odečítaní, příp. jiných variant, do jednoho kroku. Největší výhodou je přesnost výpočtu.

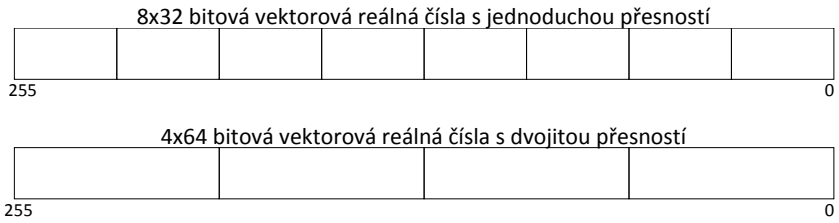
Registry YMM se překrývají s registry XMM, které tvoří dolní polovinu šířky registrů YMM, tj. bity 0 až 127. YMM registry jsou zobrazeny na následujícím obrázku B.1. Zmíněné datové typy, které přibyly k předchozím datovým typům, jsou určeny pro



Obr. B.1: 256 bitové SIMD registry YMM

práci s reálnými čísly s řádovou plovoucí čárkou a pracují na registrech YMM. Jsou to (obr. B.2)

- 8x32bitová vektorová reálná čísla s jednoduchou přesností (Single-Precision Floating-Point),
- 4x64bitová vektorová reálná čísla s dvojitou přesností (Double-Precision Floating-Point).

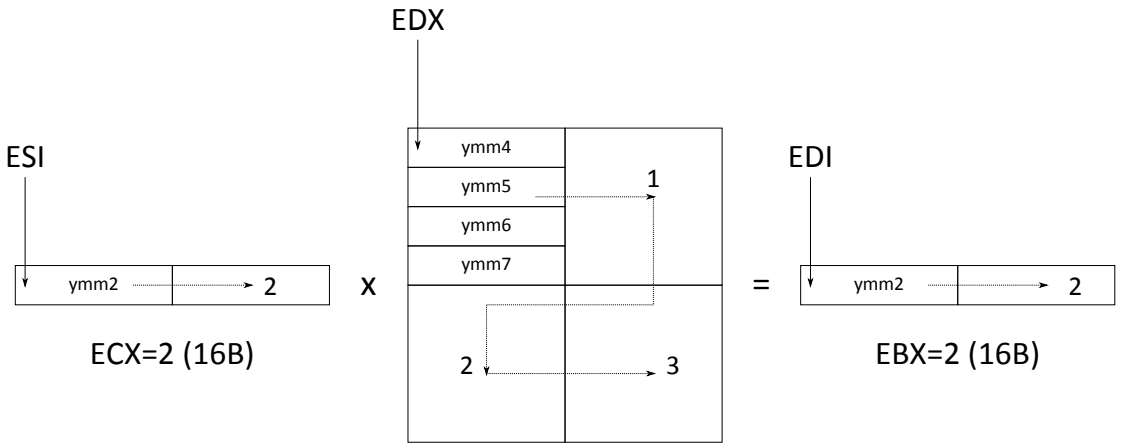


Obr. B.2: Nové datové typy technologie AVX

B.2 Vzorový příklad násobení vektoru s maticí (násobení zleva)

Oproti předchozí úloze máme k dispozici 2x větší YMM registry. Proto při použití stejné velikosti matice a vektorů můžeme pracovat s reálnými čísly s dvojitou přesností (64 bitů). Kromě YMM registrů budeme také používat standardní registry ESI, EDI, EDX, ECX a EBX, které budou plnit stejnou funkci. Za zmínku stojí, že rozměry budou nyní v násobcích 8 bytů. Také budeme využívat trojoperandový formát instrukcí.

Posuny pro matice větších rozměrů zůstanou stejné, nyní již s YMM registry.



Obr. B.3: Maticové násobení, využití registrů AVX pro větší rozměry matic

B.2.1 Zdrojové kódy

Jelikož instrukční soubory AVX, AVX2 a FMA3 poskytují nejenom předešlé instrukce, vylepšené pro práci s YMM registry, ale také zcela nové instrukce, bude

výpočet pomocí výpočetních jednotek AVX a FMA realizován zejména využitím těchto nových instrukcí.

Násobení zleva bude realizováno ve dvou variantách pro AVX a v jedné variantě pro FMA.

Výpočet - první varianta (AVX)

```
vxm_avx1:
push dword ebp
mov dword ebp,esp

sub esp,4    ;rezerva pro jednu lokální proměnnou 4B
mov dword [ebp - 4],3    ;nastavení hodnoty lokální proměnné

mov eax,32
mul dword [ebp + 24]

push eax
mul dword [ebp - 4]
mov dword [ebp - 4],eax
pop eax

mov esi,[ebp + 12]       ;ukazatel na pole x
mov ecx,[ebp + 20]    ;délka vstupního vektoru (v nas. 4B)
mov edx,[ebp + 8]        ;ukazatel na pole M


.invec7:
mov edi,[ebp + 16]       ;ukazatel na pole y
mov ebx,[ebp + 24]    ;délka výstupního vektoru (pole y)
.radmat7:
vmovupd ymm0,[edx]
add edx,eax
vmovupd ymm1,[edx]
add edx,eax
vmovupd ymm2,[edx]
add edx,eax
vmovupd ymm3,[edx]
vmovupd ymm6,[edi]
```

```
vbroadcastsd ymm7, [esi]
add esi,8
vmulpd ymm0,ymm0,ymm7
vaddpd ymm6,ymm6,ymm0

vbroadcastsd ymm7, [esi]
add esi,8
vmulpd ymm1,ymm1,ymm7
vaddpd ymm6,ymm6,ymm1

vbroadcastsd ymm7, [esi]
add esi,8
vmulpd ymm2,ymm2,ymm7
vaddpd ymm6,ymm6,ymm2

vbroadcastsd ymm7, [esi]
vmulpd ymm3,ymm3,ymm7
vaddpd ymm6,ymm6,ymm3

vmovupd [edi],ymm6

add edi,32
sub esi,24
sub edx,dword [ebp - 4]
add edx,32
dec ebx
jnz .radmat7

add esi,32
add edx,dword [ebp - 4]
dec ecx
jnz .invec7

mov dword esp,ebp
pop dword ebp
ret 40
```

Výpočet - druhá varianta (AVX)

```
vxm_avx2:
push dword ebp
mov dword ebp,esp

sub esp,4    ;rezerva pro jednu lokalni promennou 4B
mov dword [ebp - 4],3    ;nastaveni hodnoty lokalni promenne

mov eax,32
mul dword [ebp + 24]

push eax
mul dword [ebp - 4]
mov dword [ebp - 4],eax
pop eax

mov esi,[ebp + 12]    ;ukazatel na pole x
mov ecx,[ebp + 20]    ;delka vstupniho vektoru (v nas. 4B)
mov edx,[ebp + 8]    ;ukazatel na pole M

.invec8:
vmovupd ymm7,[esi]
mov edi,[ebp + 16]    ;ukazatel na pole y
mov ebx,[ebp + 24]    ;delka vystupniho vektoru (pole y)
.radmat8:
vmovupd ymm0,[edx]
add edx,eax
vmovupd ymm1,[edx]
add edx,eax
vmovupd ymm2,[edx]
add edx,eax
vmovupd ymm3,[edx]
vmovupd ymm6,[edi]
vunpcklpd ymm4,ymm7,ymm7
vperm2f128 ymm4,ymm4,ymm4,0
vmulpd ymm0, ymm0, ymm4
vaddpd ymm6, ymm6, ymm0
```

```
vunpckhpd ymm4,ymm7,ymm7
vperm2f128 ymm4,ymm4,ymm4,0
vmulpd ymm1, ymm1, ymm4
vaddpd ymm6, ymm6, ymm1

vunpcklpd ymm4,ymm7,ymm7
vperm2f128 ymm4,ymm4,ymm4,17
vmulpd ymm2, ymm2, ymm4
vaddpd ymm6, ymm6, ymm2

vunpckhpd ymm4,ymm7,ymm7
vperm2f128 ymm4,ymm4,ymm4,17
vmulpd ymm3, ymm3, ymm4
vaddpd ymm6, ymm6, ymm3
vmovupd [edi],ymm6

add edi,32
sub edx,dword [ebp - 4]
add edx,32
dec ebx
jnz .radmat8

add esi,32
add edx,dword [ebp - 4]
dec ecx
jnz .invec8

mov dword esp,ebp
pop dword ebp
ret 40
```

Výpočet - třetí varianta (FMA)

```
vxm_fma:
push dword ebp
mov dword ebp,esp
```

```
sub esp,4    ;rezerva pro jednu lokalni promennou 4B
mov dword [ebp - 4],3    ;nastaveni hodnoty lokalni promenne

mov eax,32
mul dword [ebp + 24]

push eax
mul dword [ebp - 4]
mov dword [ebp - 4],eax
pop eax

mov esi,[ebp + 12]    ;ukazatel na pole x
mov ecx,[ebp + 20]    ;delka vstupniho vektoru (v nas. 4B)
mov edx,[ebp + 8]    ;ukazatel na pole M

.invec3:
mov edi,[ebp + 16] ;ukazatel na pole y
mov ebx,[ebp + 24] ;delka vystupniho vektoru (pole y)
.radmat3:
vmovupd ymm0,[edx]
add edx,eax
vmovupd ymm1,[edx]
add edx,eax
vmovupd ymm2,[edx]
add edx,eax
vmovupd ymm3,[edx]
vmovupd ymm6,[edi]

vbroadcastsd ymm7, [esi]
add esi,8
vfmadd231pd ymm6,ymm0,ymm7

vbroadcastsd ymm7, [esi]
add esi,8
vfmadd231pd ymm6,ymm1,ymm7

vbroadcastsd ymm7, [esi]
add esi,8
```

```
vfmadd231pd ymm6,ymm2,ymm7

vbroadcastsd ymm7, [esi]
add esi,8
vfmadd231pd ymm6,ymm3,ymm7

vmovupd [edi],ymm6

add edi,32
sub esi,32
sub edx,dword [ebp - 4]
add edx,32
dec ebx
jnz .radmat3

add esi,32
add edx,dword [ebp - 4]
dec ecx
jnz .invec3

mov dword esp,ebp
pop dword ebp
ret 40
```

Opět vytvoříme dynamickou knihovnu a v skript v Matlabu, kterým budeme testovat DCT jako maticové násobení.

```
% Uzavreni vsech grafu a smazani vsech promennych
clear all; clc

f1=1000;
f2=1100;
fs=8000;

N=4096; % řád matice a vektoru
n=0:N-1;
k=0:N-1;

%definice signalu; definice v dvojité presnosti
```

```
x=0.7*cos(2*pi*f1/fs*(0:N-1))+0.3*cos(2*pi*f2/fs*(0:N-1));

%výpočet DCT verze IV; definice v dvojité presnosti
M=cos(pi/N*(n'+1/2)*(k+1/2));

disp('Výpočet v Matlabu:')
tic
x*M;
toc
y=x*M;

%vykresleni koeficientu DCT po normalizaci
plot(y * sqrt(2/N),'r') %normalizace DCT
hold on

% nacteni knihovny
hfile1 = 'test.h';
[notfound,warnings]=loadlibrary('test.dll', hfile1,'mfilename','test_mx');
pM = libpointer('doublePtr', M);
px = libpointer('doublePtr', x);

% volani funkce vxm_avx1
disp('Výpočet DCT v ASM pomocí AVX broadcast:')
td=0;
for i=1:Np
y0=zeros(1,N); py0 = libpointer('doublePtr', y0);
tic;
calllib('test', 'vxm_avx1', pM, px, py0, length(x)/4, length(y)/4);
td=td+toc/Np;
end
disp(num2str(td,6))
y_avx1=get(py0, 'Value');
plot(y_avx1 * sqrt(2/N),'g')

% volani funkce vxm_avx2
disp('Výpočet DCT v ASM pomocí AVX permutací a unpack:')
td=0;
for i=1:Np
```

```
y0=zeros(1,N); py0 = libpointer('doublePtr', y0);
tic;
calllib('test', 'vxm_avx2', pM, px, py0, length(x)/4, length(y)/4);
td=td+toc/Np;
end
disp(num2str(td,6))
y_avx2=get(py0, 'Value');
plot(y_avx2 * sqrt(2/N),'k')

% volani funkce vxm_fma
disp('Výpočet DCT v ASM pomocí FMA:')
td=0;
for i=1:Np
y0=zeros(1,N); py0 = libpointer('doublePtr', y0);
tic;
calllib('test', 'vxm_fma', pM, px, py0, length(x)/4, length(y)/4);
td=td+toc/Np;
end
disp(num2str(td,6))
y_fma=get(py0, 'Value');
plot(y_fma * sqrt(2/N),'y')
hold off
unloadlibrary test
```

B.3 Vzorový příklad násobení matice vektorem (násobení sprava)

Změna oproti násobení zleva bude opět stejná, jako v předchozí úloze. Výpočet samotný bude nyní v jedné variantě, protože instrukce tzv. horizontálního sčítání není možné při AVX použít, protože YMM registry se chovají jako dva spojené XMM registry. Tím pádem se výsledky neshodují se správnym pořadím, ale jsou přeložené.

B.3.1 Zdrojové kódy

Při výpočtě využijeme transpozici mezimatice, která vznikne po prvotním násobení. Po transpozici využijeme instrukce vektorového sčítání.

Výpočet:

```
mxv_avx1:
push dword ebp
mov dword ebp,esp

sub esp,4    ;rezerva pro jednu lokalni promennou 4B
mov dword [ebp - 4],3    ;nastaveni hodnoty lokalni promenne

mov eax,32
mul dword [ebp + 24]

push eax
mul dword [ebp - 4]
mov dword [ebp - 4],eax
pop eax

mov edi,[ebp + 16]    ;ukazatel na pole x
mov ebx,[ebp + 24]    ;delka vstupniho vektoru (v nas. 4B)
mov edx,[ebp + 8]    ;ukazatel na pole M

.invec9:
mov esi,[ebp + 12]    ;ukazatel na pole y
mov ecx,[ebp + 20]    ;delka vystupniho vektoru (pole y)
.radmat9:
vmovupd ymm0,[edx]
add edx,eax
vmovupd ymm1,[edx]
add edx,eax
vmovupd ymm2,[edx]
add edx,eax
vmovupd ymm3,[edx]
vmovupd ymm6,[edi]
vmovupd ymm7,[esi]
vmulpd ymm0,ymm0,ymm7
vmulpd ymm1,ymm1,ymm7
vmulpd ymm2,ymm2,ymm7
vmulpd ymm3,ymm3,ymm7

vunpcklpd ymm4, ymm0, ymm1
```

```
vunpckhpd ymm0, ymm0, ymm1
vunpcklpd ymm1, ymm2, ymm3
vunpckhpd ymm2, ymm2, ymm3

vperm2f128 ymm3, ymm4, ymm1, 32
vperm2f128 ymm4, ymm4, ymm1, 49
vperm2f128 ymm1, ymm0, ymm2, 32
vperm2f128 ymm2, ymm0, ymm2, 49

vaddpd ymm3,ymm3,ymm4
vaddpd ymm1,ymm1,ymm2
vaddpd ymm1,ymm1,ymm3
vaddpd ymm6,ymm6,ymm1
vmovupd [edi],ymm6

add esi,32
sub edx,dword [ebp - 4]
add edx,32
dec ecx
jnz .radmat9

add edi,32
add edx,dword [ebp - 4]
dec ebx
jnz .invec9

mov dword esp,ebp
pop dword ebp
ret 40
```

V Matlabu se skript bude téměř shodovat. Jedinou změnou bude volání jedné funkce a po definici matice a vektoru je nutné je opět oba transponovat.